THESIS

MINIMIZATION OF SOPs FOR BI-DECOMPOSABLE FUNCTIONS and NON-ORTHODOX/ORTHODOX FUNCTIONS

by

Birol Ulker

March 2002

Thesis Advisor:	Jon T. Butler
Second Reader:	Herschel H. Loomis, Jr.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Title (Mix case letters) Minimization of SOPs for Bi-decomposable functions and Non-orthodox/Orthodox functions.	**5. FUNDING NUMBERS**
6. AUTHOR(S) **ULKER, Birol**	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distrubition is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

A logical function *f* is AND bi-decomposable if it can be written as $f(X_1, X_2) = h_1(X_1)h_2(X_2)$, where X_1 and X_2 are disjoint. Such functions are important because they can be efficiently implemented. Also many benchmark functions are AND bi-decomposable. Surprisingly, the minimal sum of products (MSOP) of f is not always obtainable by finding the MSOP of h_1 and h_2 and applying the law of distributivity.

However, a special class of functions called orthodox functions, introduced by Sasao and Butler [1], do have this property. This thesis focuses on orthodox functions, and the remaining non-orthodox functions.

It is shown how to build up orthodox functions from orthodox functions on fewer variables. An algorithm is presented for generating families of non-orthodox functions. A test program is developed to test the results of the proposed algorithm and also other programs are developed to conduct experiments with both orthodox and non-orthodox functions. Results are presented that represent the first steps toward completely characterizing bi-decomposable functions that can be efficiently implemented.

14. SUBJECT TERMS Bi-decomposable functions, Orthodox functions, Non-orthodox functions, Disjoint Bi-decomposition, Minimum sum-of-products, Espresso	**15. NUMBER OF PAGES** 148
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

MINIMIZATION OF SOPs FOR BI-DECOMPOSABLE FUNCTIONS and NON-ORTHODOX/ORTHODOX FUNCTIONS

Birol Ulker
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2002

Author:	Birol Ulker
Approved by:	Jon T. Butler, Thesis Advisor
	Herschel H. Loomis, Jr., Second Reader
	Jeffrey B. Knorr, Chairman Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A logical function f is AND bi-decomposable if it can be written as $f(X_1, X_2) = h_1(X_1)h_2(X_2)$, where X_1 and X_2 are disjoint. Such functions are important because they can be efficiently implemented. Also many benchmark functions are AND bi-decomposable. Surprisingly, the minimal sum of products (MSOP) of f is not always obtainable by finding the MSOP of h_1 and h_2 and applying the law of distributivity.

However, a special class of functions called orthodox functions, introduced by Sasao and Butler [1], do have this property. This thesis focuses on orthodox functions, and the remaining non-orthodox functions.

It is shown how to build up non-orthodox functions from orthodox functions on fewer variables. An algorithm is presented for generating families of non-orthodox functions. A test program is developed to test the results of the proposed algorithm and also other programs are developed to conduct experiments with both orthodox and non-orthodox functions. Results are presented that represent the first steps toward completely characterizing bi-decomposable functions that can be efficiently implemented.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

To my thesis advisor, Prof. Butler, I would like to express my deepest thanks for all your support, guidance, confidence and patience. You were always near by me whenever I have needed you throughout my thesis work.

To my second reader, Prof. Loomis, knowing to have such a second reader like you always made me comfortable. Thank you for your assistance and various helps.

To my closest friends, Lieutenant Junior Grade Tolga Demirtas and Cihat Eryigit (sir), I will never forget your friendship and helps. You never refused to help me even in the late midnights.

To my beloved wife, Filiz, who has always stood by me, I owe you everything. I would not finish this challenging journey without you and without all your support, encouragement, and patience. You spent your nights to wait for me in front of a TV even without understanding the language. Words are not enough for me to express my appreciation.

Finally, thank you God for helping me.

EXECUTIVE SUMMARY

A logic function f is AND bi-decomposable if f can be written as $f(X,Y) = h_1(X) \wedge h_2(Y)$, where X and Y don't share any variables and $\wedge$ is the AND operation. Recently, there has been interest in such functions because many practical functions (e.g. benchmark functions) have this property [1, 6]. This interest is also inspired by the prospect that a minimum sum-of-products expression can be efficiently computed by minimizing each subfunction separately and applying the law of distributivity. That is, a divide-and-conquer algorithm can be applied that yields significantly reduced computation times.

Surprisingly, the divide-and-conquer algorithm does not always yield a minimal sum-of-products expression. Counterexamples have been shown [1, 6] where this algorithm fails. In all known counterexamples for practical problems, however, there is a small difference between the minimal sum-of-products expression and the expression obtained by the divide-and-conquer algorithm (about 4% in terms of PIs). Thus, it is an open question of whether this difference is small for all functions. Also, no characterization is known for those functions for which the divide-and-conquer algorithm fails to produce a minimal sum-of-products expression. The goal of this thesis is to address these questions.

There are functions that always yield a minimum sum-of-products expression using the divide-and-conquer algorithm. They are called *orthodox functions*, which were introduced by Sasao and Butler [1]. The importance of *orthodox* and *non-orthodox functions* is demonstrated in this thesis.

There are many minimizing tools. They try to give the best computation time and most efficient solution to logical designers. Designers usually deal with complex functions with many logical gates. Each logical gate requires a certain number of transistors depending on the technology used, and also each transistor requires a certain amount of room on chip. For example, in *cmos technology* a designer needs at least 6 transistors to build a 2-input AND gate, 6 transistors for a 2-input OR gate and 2 transistors for a 2-input NOT gate [9]. The required space for each of these gates depends

on the fabrication technology. Minimizing tools usually use the law of distributivity to provide the most improved computation time for the minimization of the logical functions, since most of the practical functions have an AND bi-decomposition property. But, as specified, not all the functions yield the minimum sum-of-products expression when the law of distributivity applied. So that, when we try to realize a logical design, we use more transistors than needed and thus more space than needed.

This thesis focuses on *non-orthodox functions* on 4 and 6-variables and *orthodox functions* on 2, 3 and 4-variables. This research may be divided into four parts.

- Determination of functions with the bi-decomposition property and the type of the bi-decomposition. To achieve this goal, several algorithms are introduced in this thesis.
- Determination of orthodox functions and characterization of their properties. This goal is accomplished by investigating known orthodox functions.
- Determination of the non-orthodox function and a characterization of their properties. To accomplish this goal, an algorithm is introduced. Functions created by this algorithm are used to explore the properties of non-orthodox functions.
- Demonstration of results observed from experiments with orthodox and non-orthodox functions.

The results obtained from this research can be divided as follows.

- The introduced algorithms and their applications.
- Results that are obtained from these algorithms.

Since orthodox and non-orthodox functions are new (they were introduced only two years ago [1]), there is not much background information and research in this area. Thus, a necessary part of this thesis work was the development of programs to investigate the new types of functions. Several algorithms and their applications are introduced. Algorithm 3, YaratNon.java, was developed to create non-orthodox functions.

SonKarar.java was developed to determine types of functions (orthodox or non-orthodox). Espresso2.java was developed to conduct logical computations between functions. Minimization tool Espresso used to minimize the functions.

This experimental research has helped to produce new lemmas, observations, and conjectures for orthodox and non-orthodox functions. They can be summarized as follows.

- Logically ANDing a non-orthodox (orthodox) function with a literal yields a non-orthodox (orthodox) function.
- Logically ORing a non-orthodox function with a literal yields a non-orthodox function.
- Logically EXORing a non-orthodox function with a literal yields a non-orthodox function.
- Complementing a non-orthodox function on 4 or 6-variables tends to produce an orthodox function (self-dual non-orthodox functions are an exception) (from experimental evidence). It appears that this does not generalize to functions with more variables.
- Logically ORing two functions on a disjoint set of variables yields a non-orthodox function if and only if at least one of the two functions are non-orthodox.
- Logically EXORing two functions on a disjoint set of variables yields a non-orthodox function if and only if at least one of the two functions are non-orthodox.
- Logically ANDing two functions on a disjoint set of variables yields a non-orthodox function if and only if at least one of the two functions are non-orthodox (from experimental evidence).

- It is shown that the counterexample that was proposed by Voight and Wegner [6], is closely related with Sasao and Butler's [1] counterexample which is the simplest known non-orthodox function.
- The penalty paid by using the law of distributivity to minimize the functions with AND bi-decomposition property, where each subfunction is non-orthodox, grows when the number of the variables of the function grows (from experimental result, it be as large 19 product terms for a 12-variable AND bi-decomposable function).
- Two representative functions are proposed. One of them has 13 don't cares and the other one has 14 don't cares. They show all 6-variable non-orthodox functions in a compact form that were discovered during the experimental research. A 6-variable non-orthodox function can be obtained from these representatives by assigning values to the don't cares. Unfortunately, not all 6-variable non-orthodox functions were discovered.

The results that are presented in this thesis represent the first steps toward completely characterizing AND bi-decomposable functions, where distributivity yields a minimal sum-of-products expression and algorithms are introduced in this thesis show a way to determine the orthodox and non-orthodox functions.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION AND BACKGROUND

A. INTRODUCTION

Minimization is one of the most important issues in logical design. When we decide to develop a circuit to perform a certain task, one of the most important steps is to create this circuit with the minimum number of logical elements, since this reduces the cost, area and latency issues. However, minimization may require too much time, affecting the circuit's cost.

There are many minimization tools. Each of them uses a different approach to solve the minimization problem. Considering the speed of the minimization process, those that split the functions into components and minimize each component separately yield the highest efficiency.

To split a function into components, the function has to be decomposable. A logic function f has a disjunctive decomposition if it can written as $f = g(h_1(X_1), h_2(X_2), ..., h_n(X_n))$, where $X_1 \cap X_2 \cap ... \cap X_n = 0$. Disjunctive decompositions can be further categorized by the number of arguments of g. For example, let g be a 2-variable AND function. Then, the decomposition property can be specified as AND bi-decomposition. Another type of decomposition is sum-of –products.

But why the sum-of-products form? Most of the simplification tools give their outputs in sum of products form, which consists of AND, OR and NOT gates. One might want to use NAND and NOR gates in the resultant function of the simplification process due to the fact that they are faster than ANDs, ORs and NOTs. The reason might be explained by help of an example. Let's consider the following logical propositions "I will not marry with you if you are not friendly or not thoughtful and also you are not handsome or not blond", "I will marry with you if you are friendly and thoughtful or if you are handsome and blond" which one sounds more natural? As we can observe from the sample expressions, using sum of products form is the most natural way to produce a logical expression. Also, we can always obtain the equivalent NANDs and NORs expressions by inserting NOTs between ANDs and ORs.

If a function has the AND bi-decomposition property, it is tempting to believe that its minimum sum of products representation can be obtained by applying the simplification processes to the components h_1, h_2 separately and applying the law of distributivity.

It is obvious that to simplify a function by applying the law of distributivity approach is less complicated and less time consuming then simplifying the function directly, but it is not true that we always have the minimum solution for the functions we want to simplify. This fact has been proved by Voight and Wegner [5] in 1989. They showed that a logical function with AND decomposition property does not necessarily give us the minimum sum of products representation when we simplify it by applying the law of distributivity.

After the first introduction of this idea, Sasao and Butler [1] took over the research. They introduced the concepts orthodox and non-orthodox functions in 2000. They showed that certain functions always give us the MSOP expression when we apply the law of distributivity. These are called *orthodox functions.* They include all 2 and 3-variable functions, unate functions, all symmetric functions, a few random functions and many benchmark functions.

The goals of this thesis are summarized as follows:

- To increase our knowledge about both orthodox and non-orthodox functions.
- To determine their behavior under certain conditions, such as after an AND operation between two orthodox and non-orthodox functions or an OR operation between a non-orthodox and an orthodox function.
- To determine other special types of functions that are non-orthodox function (Are threshold functions non-orthodox?).
- To determine an algorithm to create n-variable non-orthodox functions.
- To create an algorithm to perform experiments with the newly created functions.

To accomplish these purposes, it was necessary to consider

- Decomposition property of the functions,
- Known orthodox functions and their behaviors,
- Known non-orthodox functions and their behaviors,
- Experimental approaches to be able to determine new behaviors for both orthodox and non-orthodox functions.

The results obtained from this thesis work may be summarized as follows:

- An algorithm, Algorithm 3, is created from the studies of 4-variable non-orthodox functions. Algorithm 3 used to create n-variable non-orthodox functions. These functions used to perform experiments.
- Two algorithms are proposed. One of them is to determine the OR-type bi-decomposition and the other is EXOR-type bi-decomposition.
- By using Java language [10], a series of programs created. These programs are used in different steps of the experiments (i.e. to logically AND or OR two functions, to determine the types of the resultant functions of these logical operations as being either orthodox or non-orthodox).
- Results of the experiments that were performed with 24,576 6-variable and 512,000 8-variable functions, are used to develop new theorems, lemmas and observations for non-orthodox and orthodox functions
- Experimental results determined family representatives for some 6-variable non-orthodox functions.

B. BACKGROUND

This thesis focuses on orthodox and non-orthodox functions. Such functions are important in the context of functions that have an AND bi-decomposition. In this section, we discuss bi-decomposition and formally introduce orthodox and non-orthodox functions. Also, we discuss the *Disjoint Computation Scheme Hypothesis* [6], as it is an important part of precedent for this thesis.

Orthodox and non-orthodox functions have special meaning in the context of Disjoint Computation Scheme Hypothesis, denoted as DCSH. The main idea of this hypothesis is nothing more than the application of distributivity law. DCSH suggests that if there are two functions that have disjoint sets of variables then the AND or OR of these two functions when simplified is no better than when taking the optimal computations scheme of the two functions seperately.

It is specified in the introduction that to be able to apply this hypothesis to a function, the function has to have a property known as decomposition. This thesis narrows down its focus to the functions that have the bi-decomposition property (instead of general decomposition). So, the following sections mostly related to bi-decomposition. Besides that, since the bi-decomposition property is just a subcomponent of the decomposition property, the reader will encounter both decomposition and bi-decomposition concepts for sake of completeness.

1. Bi-decomposition

Decomposition of a Boolean function means breaking the large logic blocks into small ones, without changing the functionality of the original function. For example, the sum of products form of a function *f* is a decomposition of *f* into the OR of product terms, where each product term is the AND of variables or complements of variables. An advantage of such decomposition is the large body of knowledge and CAD tools available for their design (e.g. Espresso [8]). Another example is a fanout-free representation of a function, where no gate is allowed to drive more than one other gate. Not all functions have a fanout-free decomposition. An advantage of fanout-free representations is that they are easily tested.

Definition 1: Function *f*(*X*) has a *bi-decomposition* if it can be expressed as

$$f(X) = h_1(X_1) * h_2(X_2),$$

where * is the AND, OR or EXOR (exclusive or) on two variables.

For example function f, where $f(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4$, has OR bi-decomposition property, since we can specify $h_1(X)$ as x_1x_2 , $h_2(Y)$ as x_3x_4 and g as the OR operator.

a. Disjoint Bi-decomposition

In this section, we consider a special case of bi-decomposition.

Definition 2: Function $f(X)$ has a *disjoint bi-decomposition*, if $f(X)$ has a bi-decomposition,

$$f(X) = h_1(X_1) * h_2(X_2),$$

where X_1 and X_2 are disjoint (share no variables).

For example, same function f can be used, since it has an OR bi-decomposition property on two sets of variables $\{x_1, x_2\}$, $\{x_3, x_4\}$ that do not overlap. A function with a disjoint bi-decomposition can be realized by the circuit shown in Figure 1. Here, g represents the function *.

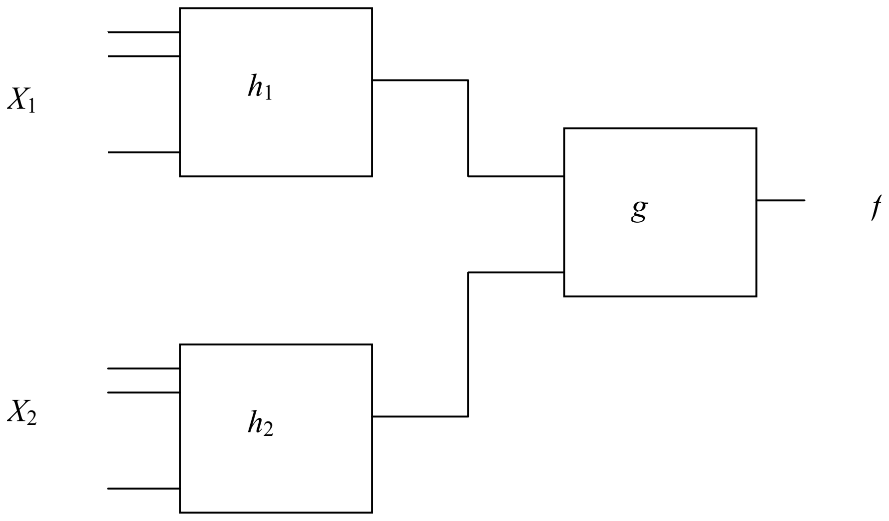


Figure 1. Disjoint Bi-decomposable function.

Definition 3: Function $f(X)$ has an *AND, OR or EXOR disjoint bi-decomposition*, if $f(X)$ has a bi-decomposition,

$$f(X) = h_1(X_1) * h_2(X_2),$$

where * is AND, OR, or exclusive OR, respectively, and X_1 and X_2 are disjoint.

Figure 1 shows the form of the circuit that realizes a function with a disjoint bi-decomposition. For example, the function in the running example has a disjoint OR bi-decomposition.

Sasao and Butler [1] have shown a synthesis technique for functions with disjoint bi-decompositions. It is known that, as the number of the variables increases, the fraction of functions with disjoint bi-decompositions becomes vanishingly small. In spite of this, the number of functions with bi-decompositions used as benchmark functions for the evaluation of logic synthesis techniques is quite large. This suggests that, although bi-decompositional functions are a small fraction of all functions, they are important in practical design applications.

b. Non-disjoint Bi-decomposition

Definition 4: A Boolean function f has a *non-disjoint bi-decomposition* if and only if f can be represented as follows,

$$f(X) = f(X_1, X_2, y) = g(h_1(X_1, y), h_2(X_2, y)),$$

where g is any 2-input logic function and $X_1 \cap X_2 = 0$ and y is a single variable.

Figure 2 shows the circuit realization of a function with a non-disjoint bi-decomposition. In the case of non-disjoint bi-decompositions, it is more difficult to find the component functions than in the case of disjoint bi-decompositions.

Consider the function f, where $f(x_1, x_2, x_3) = \overline{x_1}x_3 + x_1x_2$. The function f can be represented as $f(X) = f(X_1, X_2, x_1) = g(h_1(X_1, x_1), h_2(X_2, x_1))$, where $X_1 \cap X_2 = 0$ and g is OR operand so that f has an OR bi-decomposition. This function also can also be represented as follows; $f(x_1, x_2, x_3) = \overline{x_1}x_3 \oplus x_1x_2$ and $f(x_1, x_2, x_3) = (\overline{x_1} + x_3)(x_1 + x_2)$. Thus, f has both OR and EXOR bi-decomposition [1].

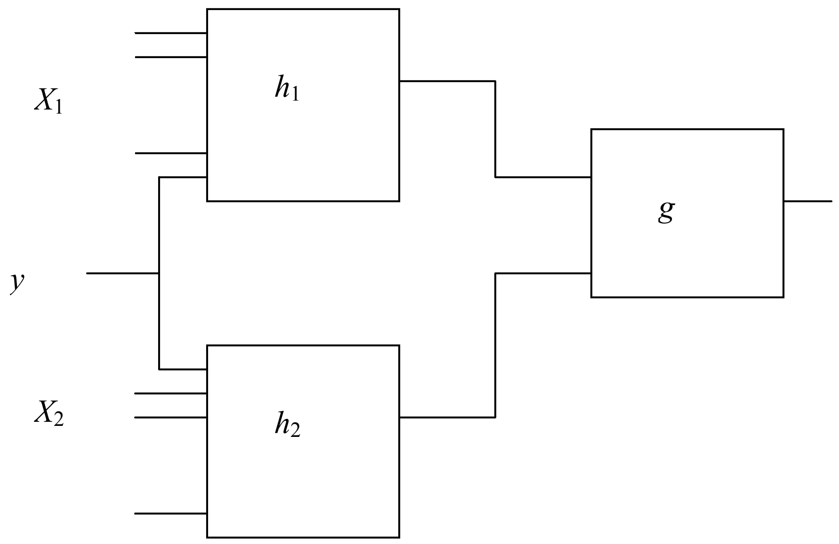


Figure 2. Non-disjoint Bi-decomposable function

c. Methods to Determine the Type of Disjoint Bi-decomposition

In this section, we show how to determine the component functions of the functions that have an OR or EXOR disjoint decomposition. This is sufficient for finding the component functions of a function with any kind of disjoint decomposition, since a function f has an AND disjoint bi-decomposition if and only if $\overline{f}$ has the OR disjoint bi-decomposition.

1. Algorithm for functions with an OR-type bi-decomposition. A function f has the OR disjoint bi-decomposition property, $f(X_1, X_2) = h_1(X_1)$ OR $h_2(X_2)$, if and only if every product term of ISOP (irredundant sum-of-products) consists of literals belonging to input set X_1 only or X_2 only. We can use several methods to determine if a function has an OR bi-decomposition property or not;

- Using the Karnaugh map

The function should have all 1's grouped in a subset of columns and a subset of rows and also none of these columns and rows can contain 0's [7].

cd \ ab	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	1	0	1	0
10	1	1	1	1

Figure 3. Karnaugh map of an OR bi-decomposable function

Figure 3 shows a function *f*, $f(a,b,c,d) = \overline{a} \oplus b + c\overline{d}$, that has the OR bi-decomposition. In this case, f can be represented as $f(X_1, X_2) = h_1(X_1) + h_2(X_2)$ where, $h_1(X_1) = \overline{a} \oplus b, h_2(X_2) = c\overline{d}$, $X_1 = \{a,b\}$ and $X_2 = \{c,d\}$. This method is useful for functions that have a small number of inputs.

- Using the ISOP representations of the functions

The *ISOP* representation of a function is the OR of prime implicants (PIs), none of which are redundant. Application of this method relies on examining the PIs with respect to common literals and creating subsets with the literals of the PIs that have at least one common literal.

Let $PI = \{P_1, P_2, ..., P_m\}$ be the set of PIs associated with the given ISOP. The algorithm for determining the OR bi-decomposition of *f* forms disjoint subsets $X_1^{'}, X_2^{'}, ..., X_p^{'}$ of *X*, the set of variables. The algorithm examines each P_i, in turn modifying $X_1^{'}, X_2^{'}$, etc. as it proceeds.

```
ALGORITHM 1
1. $X_1 \leftarrow \phi$
2. for $i = 0$ to $m$ do
        if ($P_i$ shares a variable with any $X_j^{'}$)
        coalesce into one subset all that share a variable with $P_i$.
        else
        form a new subset containing all variables in $P_i$.
   end
3. if (there is one subset)
        stop (failure)
   else
        $h_1(X) \leftarrow$ OR of all a $P_i$'s such that $P_i$ depends on variables in $X_1^{'}$.
        $h_2(X) \leftarrow$ OR of all other $P_i$'s.
stop (success).
```

For example, apply the algorithm to the function $f(a,b,c,d) = ab + bc + de$. Let $P_1 = ab$, $P_2 = bc$ and $P_3 = de$. Create the first subset with the literals of P_1 call it X_1, $X_1 = \{a,b\}$. Examine P_2 with respect to X_1. Since they share the literal b, add the literals of P_2 to subset X_1. Proceed by examining the product term P_3 with respect to subset X_1. Since it does not have common literals with subset X_1, create a new subset called X_2. X_2 consists of the literals of P_3, i.e. $X_2 = \{d,e\}$. Since the total number of the subsets is 2, function *f* has an OR bi-decomposition, and it can be written as $f(X_1, X_2) = h_1(X_1) + h_2(X_2)$, where $X_1 = \{a,b,c\}$, $X_2 = \{d,e\}, h_1(X_1) = ab + bc$, and $h_2(X_2) = de$.

This method can also be used for the determination of the OR decomposition property. As it is specified earlier in the discussion, bi-decomposition is a subset of decomposition that requires exactly two subfunctions, while decomposition has more subfunctions.

2. Exor type bi-decomposition algorithms. Among the different ways of representing an arbitrary function is the Reed-Muller expression. In this representation, we have a standard expression for the representation of the all n-variable functions, called the *positive polarity Reed-Muller expression* (PPRM). It is formed as follows;

$$f(x_1, x_2, ..., x_n) = a_0 \oplus (a_1 x_1 \oplus a_2 x_2 \oplus ... \oplus a_n x_n) \oplus (a_{12} x_1 x_2 \oplus a_{13} x_1 x_3$$

$\oplus ... \oplus a_{n...1} x_{n...1} x_n) \oplus ... \oplus a_{12...n} x_1 x_2 ... x_n$, where $a_i \in \{0,1\}$. For a given function *f*, the coefficients are uniquely determined.

A given function *f* has the EXOR bi-decomposition property, $f(X_1, X_2) = h_1(X_1)\ EXOR\ h_2(X_2)$, if and only if every product in the PPRM for *f* consists of literals that belong to set X_1 only or X_2 only. As in the OR bi-decomposition case, we have different ways to determine whether a function has this type of bi-decmposition.

- Using the Reed-Muller expression

Like ISOP's, all implicants in a Reed-Muller PPRM are irredundant. As in the usage of the ISOP representation for determining the OR bi-decomposition property, this method also relies the examination of the product terms with respect to common literals and creating subsets with the literals of the product terms that have at least one common literal. At the end of the process, we should have at least two subsets of literals to be able to state that the function under examination has the EXOR decomposition property. If we have 2 or more subsets of disjoint variables, then the function has the EXOR bi-decomposition property.

Let $x_i, i = (1,2,...,n)$, be the input variables of *f*. Let $p_1 \oplus p_2 \oplus ... \oplus p_t$ be the PPRM for function *f*, where $p_i\ (i = 1,2,...,t)$ are products. The algorithm forms disjoint subsets $X_1^{'}, X_2^{'}, ..., X_p^{'}$ of *X*, the set of variables. The algorithm examines each P_i, in turn modifying $X_1^{'}, X_2^{'}$, etc. as it proceeds.

```
ALGORITHM 2
1. $X_1 \leftarrow \phi$
2. for $i$ = 0 to $t$ do
        if ($p_i$ shares a variable with any $X_j^{'}$)
            coalesce into one subset all that share a variable with $P_i$.
        else
            form a new subset containing all variables in $p_i$.
   end
3. if (there is one subset)
        stop (failure)
   else
        $h_1(X) \leftarrow$ EXOR of all a $p_i$'s such that $p_i$ depends on variables in $X_1^{'}$.
        $h_2(X) \leftarrow$ EXOR of all other $p_i$'s.
   stop (success).
```

For example, apply the algorithm to the function *f*, $f(a,b,c,d,e) = ab \oplus cd \oplus ae$. Let P_1= *ab,* P_2=*bc* and P_3=*de.* Create the first subset with the literals of the P_1, call it X_1, where $X_1 = \{a,b\}$. Examine P_2 with respect to X_1 since they do not share any literals create a new subset called X_2. X_2 consists of the literals of P_2, $X_2 = \{c,d\}$. Continue examining the product terms with P_3, examine it with respect to X_1 first, since it does have a common literal with subset X_1, namely *b*, add the literals of the P_3 to subset X_1. Now we're done with the examination of the product terms and we have two subsets X_1 and X_2. Since the number of the subsets is more than one we can state that function *f* has the EXOR bi-decomposition property and we can represent it as following $f(X_1, X_2) = h_1(X_1) \oplus h_2(X_2)$ where $X_1 = \{a,b,e\}, X_2 = \{c,d\}$, $h_1(X_1) = ab \oplus ae$ and $h_2(X_2) = cd$.

d. Number of the Functions with Bi-decomposition Property

As specified earlier, there are two types of bi-decompositions; disjoint and non-disjoint. In the previous sections, different algorithms are presented to find various bi-decompositions of a function. In the case of disjoint bi-decomposition, it is easy to determine the type of bi-decomposition. But, this is not the case for the non-disjoint bi-decomposition.

Before presenting the numerical results consider the following.

Definition: Two functions, f_1 and f_2, are NP-equivalent if f_2 can be obtained from f_1 by complementing and/or permuting variables of f_1. For example, $f_1 = x_1 + \overline{x_2}$ is NP-equivalent to $f_2 = \overline{x_1} + x_2$, since f_2 is obtained from f_1 by interchanging (permuting) x_2 and x_1.

All 2-variable functions have a bi-decomposition as shown in Table 1. Consider the function $f(x_1, x_2) = x_1$, it is a function that depends on 1 of the 2 variables. This function has the AND bi-decomposition property, since it can be represented as $f(X_1, X_2) = h_1(X_1)h_2(X_2)$, where $h_1(X_1) = x_1$ and $h_2(X_2) = 1$. It also has an OR bi-decomposition, since it can be expressed as $f(X_1, X_2) = h_1(X_1) \vee h_2(X_2)$ where $h_1(X_1) = x_1$ and $h_2(X_2) = 0$. Similarly, it has an EXOR bi-decomposition.

2-variable function	# of variables function depend on	Type of bi-decomposition
0	0	
1	0	
x_1	1	AND bi-decomposition
x_2	1	AND bi-decomposition
$\overline{x_1}$	1	AND bi-decomposition
$\overline{x_2}$	1	AND bi-decomposition
x_1x_2	2	AND bi-decomposition
$x_1\overline{x_2}$	2	AND bi-decomposition
$\overline{x}_1x_2$	2	AND bi-decomposition
$\overline{x}_1\overline{x_2}$	2	AND bi-decomposition
$x_1 + x_2$	2	OR bi-decomposition
$x_1 + \overline{x_2}$	2	OR bi-decomposition
$\overline{x_1} + x_2$	2	OR bi-decomposition
$\overline{x_1} + \overline{x_2}$	2	OR bi-decomposition
$x_1 \oplus x_2$	2	EXOR bi-decomposition
$x_1 \oplus \overline{x_2}$	2	EXOR bi-decomposition

Table 1. All 2-variable functions.

In the case of three variable functions, we have a total of 256 logical functions and these functions can divided into 16 NP-equivalence classes as shown in the Table2 [3].

Representative function of the NP-equivalence class	# of functions	Type of the bi-decomposition	Property of the bi-decomposition
$x_1 \oplus x_2 \oplus x_3$	2	EXOR	DISJOINT BI-DECOMPOSITION
$x_1x_2x_3$	8	AND	
$x_1 + x_2 + x_3$	8	OR	
$x_1 (x_2 + x_3)$	24	AND	
$x_1 + (x_2 x_3)$	24	OR	
$x_1 (x_2 \oplus x_3)$	12	AND	
$x_1 + (x_2 \oplus x_3)$	12	OR	
$x_1 \oplus x_2 x_3$	24	EXOR	
$x_1x_2x_3 + \bar{x}_1 \bar{x}_2 \bar{x}_3$	4		NON-DISJOINT BI-DECOMPOSITION (one variable common)
$(x_1+x_2+x_3)(\bar{x}_1+\bar{x}_2+\bar{x}_3)$	4		
$\bar{x}_1x_3 + x_1x_2$	24		
$x_1 \bar{x}_2 \bar{x}_3 + x_2x_3$	24		
$(\bar{x}_1+\bar{x}_2+\bar{x}_3)(x_2+x_3)$	24		
$x_1x_2 + x_2x_3 + x_3x_1$	8		NO NON-DISJOINT BI-DECOMPOSITION (one variable common)
$x_1x_2 + x_2x_3 + x_1x_3 + \bar{x}_1 \bar{x}_2 \bar{x}_3$	8		
$\bar{x}_1x_2x_3 + x_1 \bar{x}_2x_3 + x_1x_2 \bar{x}_3$	8		

Table 2. NP-equivalence representation of three variable functions.

Table 3 [3] shows the number of the NP-equivalence classes and type of the bi-decomposition that each class has for functions with two, three and four variables as a summary.

Number of the variables		n = 2	n = 3	n = 4
Number of the functions		16	256	65536
Number of functions with Disjoint Bi-decomposition	AND	4	44	1660
	OR	4	44	1660
	EXOR	2	26	914
Number of functions with Non-disjoint Bi-decomposition		0	80	3680
TOTAL		10	194	8094

Table 3. Summary of the two, three and four variables functions.

2. Espresso

The program Espresso is important in this thesis, since all the lemmas, conjectures and theorems presented in Chapter V were inspired by the results of this software [8]. It is discussed in detail here.

The most important portion of this thesis is the experimental research part, and for conducting the experiments, a tool is needed that could minimize the functions under test to their minimum sum of products in a reliable way and also with acceptable speed, since the number of the functions that was planned to be dealt with was too large. In respect to these conditions, available tools were evaluated and eventually Espresso was chosen.

The primary reasons for choosing Espresso are

- More reliable results
- Short process time
- Free
- Available both for windows and unix environment

- Robustness (unlike other tools, it was developed by not a person but a big team from University of California- Berkeley so that it is more robust.)

Espresso version 2.3 was released by Berkeley on 31 January 1988. It is based on the *Quine-McCluskey Method*, which simplifies a logical expression that is in disjunctive normal form, to obtain an equivalent minimal disjunction of conjunctions (sum of products).

Since Espresso is based on the Quine-McCluskey Method, Quine-McCluskey Method will be discussed briefly. The Quine-McCluskey Method is based on repeated applications of the distributive law and the complement law ($a \vee \bar{a} = 1$).

For example, consider $xyz + x\bar{y}z = x(y + \bar{y})z = xz$.

The steps of the algorithm can be summarized as *rewrite, reduction and selection step*.

Rewrite step: The first step in the minimization process according to Quine-McCluskey method is to rewrite the minterms using 1s and 0s instead of the literals. This new representation is called the bit string form [8].

Reduction step: In this step of the Quine-McCluskey method, pairs of the strings are compared. If two bit strings agree in all bits and disagree in one bit (e.g. 111 and 101), they are combined. A table is formed by usage of the generated bit strings called reduction table [8].

Selection step: After the accomplishment of reduction step, a table (selection table) is formed. It has the original minterms as column headers. The reduction table is examined to take the term with the fewest literals. This term becomes the first row of selection table. The minterms that are used to obtain this term are marked by asterisks. The minterms that are not labeled become the missing minterms and the terms from the reduction table that have this/these missing minterm or minterms become the next rows of the selection table [8].

a. Keywords and Usage of Espresso

Espresso takes a 2-level representation of a two-valued or multi-valued logical expression and produces a minimal equivalent representation for this function. Also, it automatically verifies that the minimal representation obtained at the end of the process is equivalent to the original function.

Espresso reads the provided file, performs the minimization, and outputs the result as a file (or it can prompt the result directly to the screen of the computer). The user can provide the input function in different ways: he/she can use the ON-set representation, ON-set and DC-set, ON-set and OFF-set or ON-set, OFF-set and DC-set. ON-set refers to the minterms that imply the function value is a 1. OFF-set refers the minterms that imply the function value is a 0. DC-set refers the minterms that are unspecified, namely don't cares. The default for Espresso is the ON-set. Figure 4 illustrates an example input file (for the function $f(x,y,z,w) = \overline{x}\,\overline{y}z\overline{w} + \overline{x}\,\overline{y}zw + \overline{x}y\overline{z}\,\overline{w} + \overline{x}y\overline{z}w + x\overline{y}\,\overline{z}\,\overline{w} + x\overline{y}\,\overline{z}w + \overline{x}yzw + x\overline{y}zw + xy\overline{z}w + xyzw$).

```
.i 4
.o 1
0010 1
0011 1
0100 1
0101 1
1000 1
1001 1
0111 1
1011 1
1101 1
1111 1
.e
```

Figure 4. An ON-set input file for Espresso.

Figure 4 shows a very basic input file, which is sufficient for the minimization of the function. If we go through the input file, we encounter keywords ".i", ".e" and ".o".

- **.i (number):** Specifies the number of the input variables. In our running example, it is 4 and thus the usage is ".i 4".
- **.o (number):** Specifies the number of the output functions. In our running example, it is 1, and thus the usage is ".o 1".
- **.e :** Marks the end of the description of the product terms.

Certain lines between the keywords .i and .e represent product terms of the function. 0 stands for a complemented literal (i.e. $\overline{x_i}$) of the minterm and 1 stands for a not complemented (i.e. x_i).

Also, we can have comment lines, which do not have any effect in the minimization process. These lines begin with the pound sign (#) into the line. These of course are not the only keywords of the tool, but for our purpose, these are sufficient. Figure 5 shows the output file of Espresso corresponding to the input file in Figure 4.

```
.i 4
.o 1
.p 5
-1-1 1
--11 1
100- 1
010- 1
001- 1
.e
```

Figure 5. Typical output file of Espresso.

As we can see in Figure 5, we have a new keyword, which is ".p". This keyword specifies the number of the PIs in the minimized sum-of-products expression of the function. Since this output file belongs to our running example function *f*, we can state that the MSOP for function *f* has 5 PIs. So the minimized equivalent of our running example is $f(x, y, z, w) = \bar{x}y\bar{z} + x\bar{y}\bar{z} + \bar{x}\bar{y}z + yw + zw$.

To invoke Espresso use the format *espresso (inputfile-name).es* > *(outputfile-name).out*. As can be seen from the syntax, we use the ".es" extension for the input files and ".out" extensions for the output files. Since this tool works in the Unix operating system to create an input file, one can use either *texteditor* or *vi* editor of Unix.

Besides the keywords, Espresso has flags. One that was especially useful in the research presented in this thesis is –Dexact. For 10 or more variable functions flag -Dexact guarantees the minimum number of product terms at the end of the minimization process. Also experiments showed that usage of this flag might cause excessive computation time, particularly with 12 and more variable functions. In Chapter V, a table will show the average computation times of Espresso for functions with different number of variables.

After the introduction in this chapter of disjoint bi-decomposition and the minimization tool, Espresso, notations and basic definitions are addressed in Chapter II of this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

II. NOTATION AND DEFINITIONS

A. LITERAL

Let *y* be a switching variable; i.e. $y \in \{0,1\}$ and let $\overline{y}$ be the complement of *y.* The logical AND of two or more literals is a *product* or *implicant*. Product *p* is an implicant of function *f*, if *f* is 1 whenever *p* is 1 [2].

Consider the following example; Let $f(X) = f(x_1, x_2, x_3) = x_1x_2 + \overline{x_2}x_3$. It is clear that x_1x_2 and $\overline{x_2}x_3$ are implicants of $f(X)$. However, so also is x_1x_3 since $f(X)$ is 1 whenever x_1x_3 is 1. Further, $\overline{x_1}x_2$ is not an implicant of $f(X)$, since $f = 0$ and $\overline{x_1}x_2 = 1$ when $x_1x_2x_3 = 010$.

B. PRIME IMPLICANT (PI)

Implicant p is a *prime implicant* of $f(X)$ if the elimination of one or more literals causes p not to be an implicant of $f(X)$.

A prime implicant of $f(X)$ is an *essential prime implicant* if there exists an input a, where $p(a) = 1$ but $p^{'}(a) = 0$ for all other prime implicants of $f(X)$.

Consider, $f(x_1, x_2, x_3) = x_2 + \overline{x_1}x_3$, Figure 6 shows the minterms and the prime implicants of the function *f.* There are two PIs; $p_1 = x_2$ and $p_2 = \overline{x_1}x_3$.

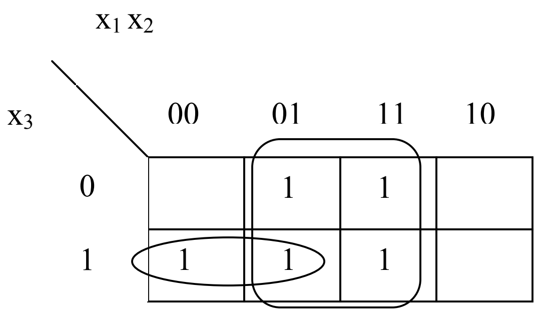


Figure 6. Function *f*, $f(x_1, x_2, x_3) = x_2 + \overline{x_1}x_3$.

C. MINTERM AND MAXTERM

A minterm is a product term with *n* literals such that each variable appears exactly once. On the other hand, a *maxterm* is a sum term with *n* literals such that each variable appears exactly once [4]. Table 4 shows minterms and maxterms for a 3-variable logic function, where variable set $X = \{x, y, z\}$.

Minterms also can be named as *ON-set* and maxterms as *OFF-set*.

x y z	f	MINTERMS	MAXTERMS
0 0 0	$f(0,0,0)$	$\bar{x}\bar{y}\bar{z}$	$x+y+z$
0 0 1	$f(0,0,1)$	$\bar{x}\bar{y}z$	$x+y+\bar{z}$
0 1 0	$f(0,1,0)$	$\bar{x}y\bar{z}$	$x+\bar{y}+z$
0 1 1	$f(0,1,1)$	$\bar{x}yz$	$x+\bar{y}+\bar{z}$
1 0 0	$f(1,0,0)$	$x\bar{y}\bar{z}$	$\bar{x}+y+z$
1 0 1	$f(1,0,1)$	$x\bar{y}z$	$\bar{x}+y+\bar{z}$
1 1 0	$f(1,1,0)$	$xy\bar{z}$	$\bar{x}+\bar{y}+z$
1 1 1	$f(1,1,1)$	xyz	$\bar{x}+\bar{y}+\bar{z}$

Table 4. Minterms and Maxterms for 3-variable logic function *f*(*x,y,z*).

D. CUBE NOTATION

A product term can be expressed in *cube notation* as follows. Each occurrence of x_i in *p* is represented by a 1. Each occurrence of $\overline{x_i}$ is represented by a 0. Missing variables are represented by a -.

For example, the two 4-variable minterms $\overline{x_1}x_2\overline{x_3}\,\overline{x_4}$ and $x_1x_2\overline{x_3}\,\overline{x_4}$ have the cube representations 0100 and 1100, respectively. Note that these two minterms combine together into a single product term, as follows.

$$x_1 x_2 \overline{x_3}\,\overline{x_4} \vee \overline{x_1} x_2 \overline{x_3}\,\overline{x_4} = x_2 \overline{x_3}\,\overline{x_4} .$$

The resultant product term, $x_2 \overline{x_3}\,\overline{x_4}$ has the cube notation -100.

E. STRONG AND WEAK MINTERMS

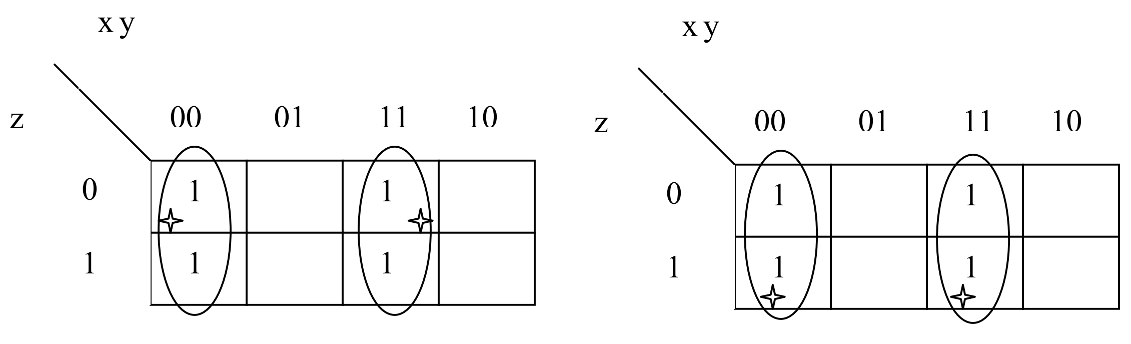
Let *m* be a minterm of *f*. If the number of the 1's in m is $\frac{n}{2}$ or more (or less), then *m* is a *strong* (*weak*) *minterm*, where *n* is the number of variables of *f*.

For example, the minterms 01000 and 11010 are weak and strong minterms, respectively. Minterm 001011 is both strong and weak.

F. INDEPENDENT SET OF MINTERMS

Let $M(f)$ be the set of true minterms for function *f*. Then, $MI(f) \subseteq M(f)$ is an *independent set of minterms* of function *f*, if and only if no PI of f covers more than one minterm in $MI(f)$. $\eta(f)$ is the number of the minterms in $MI(f)$; i.e. $\eta(f) = |MI(f)|$.

Consider the Karnaugh map representation of function *f*, $f(x,y,z) = \overline{x}\,\overline{y} + xy$, in Figure 7. This function has 4 different maximal independent sets of minterms; $\{000,011\}$, $\{001,111\}$, $\{000,111\}$ and $\{001,110\}$. Minterms marked by stars in Figure1.4 shows two sets. Note that $\eta(f) = 2$.



(a) (b)

Figure 7. (a) Shows the first MIS (b) Shows the second MIS.

G. DISTINGUISHED MINTERM

Given a function *f*, let $M(f)$ be the set of true minterms of *f*. Then, $MD(f) \subseteq M(f)$ is a set of *distinguished minterms*, if exactly one PI of f covers each minterm in $MD(f)$ [1].

Consider the function *f*, where function $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$. This function has 3 distinguished minterms, namely 010, 100 and 001. Figure 8 shows the Karnaugh map representation of the function *f* and the distinguished minterms of it.

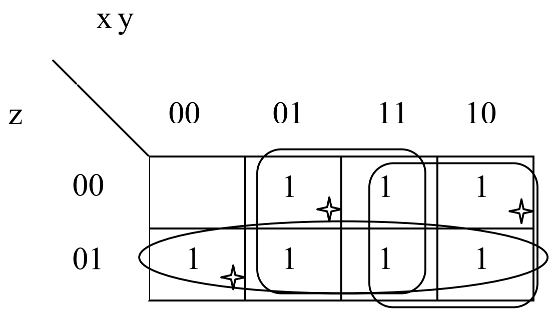


Figure 8. Karnaugh map representation of the function *f*, minterms that denoted with ✧ are the distinguished minterms.

Note that a set of distinguished minterms is an independent set of minterms. The converse is not true. Figure 9 shows a three variable function whose maximal independent set has three minterms (marked by stars). However this set is not a set of distinguished minterms.

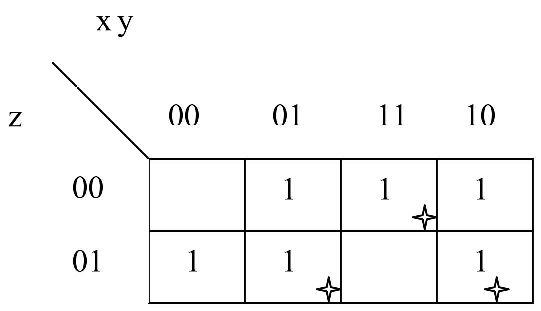


Figure 9. Maximal independent set is not always the set of Distinguished minterms.

H. ISOP (IRREDUNDANT SUM OF PRODUCTS)

The logical OR of all products (implicants) of a function is the complete sum-of-product (CSOP) of a function *f*. If we eliminate products from the CSOP of *f* to the point where eliminating any remaining products will change the function yields an *irredundant sum of products* (ISOP).

I. MSOP (MINIMUM SUM OF PRODUCTS)

Among all the ISOPs, one that has the fewest PIs is called a *minimum sum of products* (MSOP). The cost of function *f*, $\tau(MSOP : f)$ is the number of the prime implicants in the MSOP of a given function *f*, e.g. $\tau(MSOP : f) = 2$ in Figure 6.

J. SYMMETRIC FUNCTION

A function *f* is *symmetric* in variables x_i and x_j if interchanging x_i and x_j leaves *f* unchanged. For example, both $x_1x_2x_3$ and $x_1x_2 + x_3$ are symmetric in x_1 and x_2. A function is *symmetric*, if it is symmetric in all pairs of variables [2].

S_A^n denotes a *symmetric function*, that has the logical value 1 if *m* of its *n* variables are 1, where $m \in A$, and has the logical value 0 otherwise.

Example; symmetric function $S_{1,3}(x, y, z)$ can be written as follows, $f(x, y, z) = xyz + \overline{x}\overline{y}z + x\overline{y}\overline{z} + \overline{x}y\overline{z}$. *f* is 1 when 1 or 3 of its variables are 1, and is 0 otherwise.

K. UNATE FUNCTION

A function $f(x_1, x_2, ..., x_n)$ is positive in variable x_i, if there exists a SOP (conjunctive expression) for the function in which x_i appears only in uncomplemented form. $f(x_1, x_2, ..., x_n)$ is negative in x_i, if there exists a SOP (conjunctive expression) for the function in which x_i appears only in complemented form. If *f* is either positive or negative in x_i, then it is said to be *unate* in x_i. If a function is unate in each of its variables, then this function called a *unate function* [2].

Consider the function *f*, $f(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4$, which is an unate function, since it is positive in all its variables. Figure 10 shows the Karnaugh map representation of function *f*.

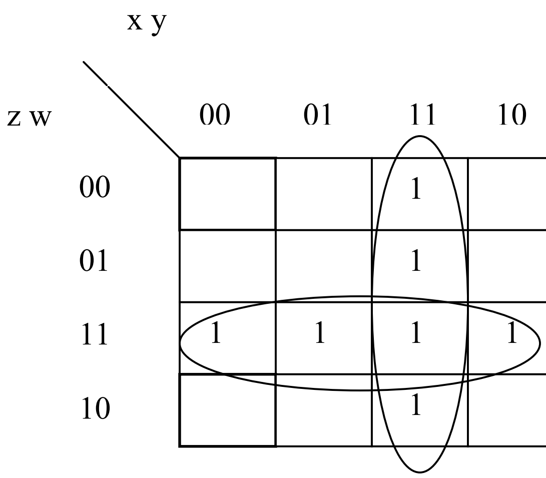


Figure 10. Karnaugh map representation of a unite function *f*.

L. MAJORITY FUNCTION

Majority functions are a subclass of symmetric functions. A symmetric function *f* is a *majority function* if and only if, the number of the variables is odd and *f* is 1 if and only if more than half of the variables are 1 [2].

The symmetric function *f*, $f_2(x_1, x_2, x_3) = x_1x_2 + x_2x_3 + x_1x_3$ is a majority function, since 2 of the 3 variables must be 1 (true) for the whole function to be 1 (true). Figure 11 shows the Karnaugh map representation of the function.

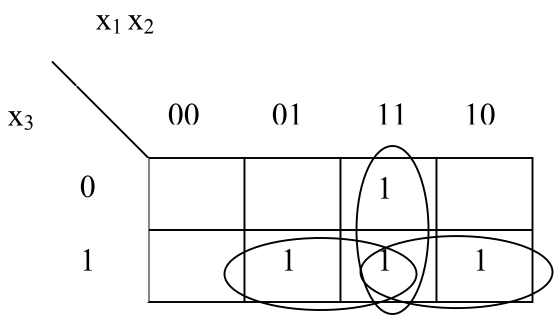


Figure 11. Minterms of a majority function.

M. SELF DUAL FUNCTION

An arbitrary function f is said to be a *self dual function* if and only if $f(x_1, x_2, x_3, ..., x_n) = \overline{f}(\overline{x_1}, \overline{x_2}, \overline{x_3}, ..., \overline{x_n})$ [2].

Consider the function $f(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2x_3$. It is self dual. Since, this function is the symmetric function $S_{2,3}(x_1, x_2, x_3)$, which is 1 if and only if 2 or 3 of the variables are 1, complementing the variables of $S_{2,3}(x_1, x_2, x_3)$ yields a function that is 1 if 0 or 1 of the variables are 1. But, this is also the complement of $S_{2,3}(x_1, x_2, x_3)$.

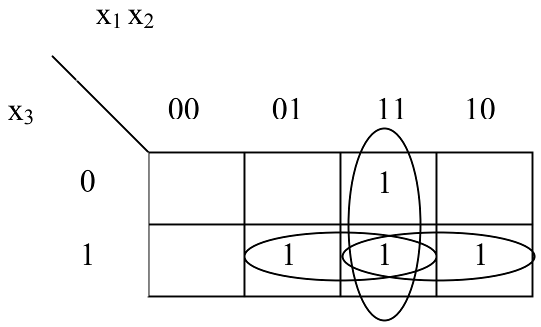


Figure 12. Karnaugh map representation of Self dual function *f*.

N. INCOMPLETELY AND COMPLETELY SPECIFIED FUNCTIONS

Let f be an *incompletely specified* symmetric function on n-variables given as follows.

$f(x_1, x_2, ..., x_n)$ =0 if all variables are 0

=1 if one or zero variables are 0

= - (don't care) otherwise, where $n > 2$,

Figure 13 shows this function for the case n=3.

A *completely specified* function g is said to cover an incompletely specified function f if g is 0 and 1 for all assignments of values to variables for which f is 0 and 1, respectively [1].

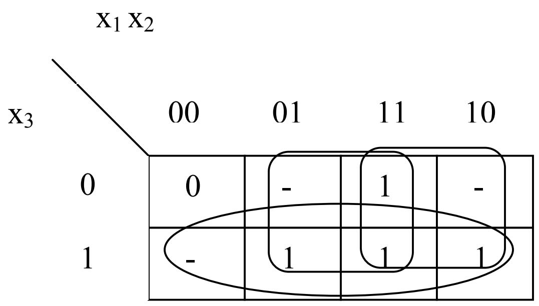


Figure 13. An incompletely specified function on 3-variables.

O. NP-EQUIVALENT

Consider two functions; $f(X)$ and $g(X)$. Function $f(X)$ is NP-equivalent to $g(X)$ if, $g(X)$ can be obtained by a complementation and/or permutation of the variables of $f(X)$.

Consider the functions $f(X) = x_1 + \overline{x_2}$ and $g(X) = \overline{x_1} + x_2$, where $X = \{x_1, x_2\}$. Function $f(X)$ is NP-equivalent to $g(X)$, since it is obtained from the permutation of the variables of function $f(X)$. Also note that $f(X)$ can be obtained from $g(X)$ by a complementation of variables.

P. CONCLUSIONS OF THE CHAPTER

In this chapter the notations and basic definitions that are going to be mostly encountered by the reader presented. The orthodox functions and their known properties are addressed in the following chapter, Chapter III.

III. ORTHODOX FUNCTIONS

This chapter focuses on a special type of function called an *orthodox function*. Orthodox functions are important because, if the subfunctions of an AND bi-decomposable function *f* are orthodox functions, then a circuit for *f* can be designed by using a divide-and-conquer algorithm that dramatically reduces the design cost.

Before introducing orthodox functions we discuss the *Disjoint Computation Scheme Hypothesis,* or DCSH proposed by Voight and Wegner [5].

A. DISJOINT COMPUTATION SCHEME HYPOTHESIS (DCSH)

The Direct conjunction or the AND of two functions, where the variable subsets are disjoint from each other, can be defined as follows $(f \wedge g)(x_1, x_2, ..., x_n, y_1, y_2, ...y_n) =$ $(f(x_1, x_2, ..., x_n) \wedge g(y_1, y_2, ..., y_n))$ and direct disjunction or the OR of two functions, where the variables subsets are disjoint from each other, can be defined as follows; $(f \vee g)(x_1, x_2, ..., x_n, y_1, y_2, ..., y_n) = (f(x_1, x_2, ..., x_n) \vee g(y_1, y_2, ..., y_n))$.

One might expect that the minimal sum-of products expression of f^g or $f \vee g$ can be obtained by finding the minimal sum-of-products expression *f* and *g* separately and computing the conjunction or disjunction of them. The Disjoint Computation Scheme Hypothesis states this. In applying it, we achieve a significant advantage since the computation time for computing the MSOP of *f* and *g* is usually much less than for computing the MSOP of f^g or $f \vee g$.

Lemma; let p_1 and p_2 be implicants on X_1 and X_2, where $X_1 \cap X_2 = 0$. Products p_1 and p_2 are PI's of $h_1(X_1)$ and $h_2(X_2)$ respectively if and only if;

- p_1 and p_2 are PI's of $h_1(X_1) \vee h_2(X_2)$, and
- $p_1\ p_2$ is a PI of $h_1(X_1)h_2(X_2)$

Following is the proof of the above statement. If p is a PI of either h_1 or h_2, it is trivially an implicant of $h_1 \vee h_2$. Since $X_1 \cap X_2 = 0$, p is also a PI of $h_1 \vee h_2$. Let p be a PI of $h_1 \vee h_2$. It can be expressed as $p = p_1 p_2$, where p_1 consists of literals from X_1 only and p_2 consists of literals from X_2 only. Since p is a PI of $h_1 \vee h_2$, an assignment of values to the variables associated with $p_1 p_2$ causes either h_1 or h_2 or both to be 1. Assume h_1 is 1. Since h_1 is 1, p_1 is an implicant of h_1. But, $p_1 p_2$ can't be a PI unless $p_2 = 1$. p_1 must be a PI of h_1. On the contrary, if not, it implies a PI, $p_1^{'}$ of h_1. Thus $p_1^{'} p_2$ must be a product that implies $h_1 \vee h_2$, that is implied by $p_1 p_2$. But this is a contradiction, since $p_1 p_2$ is a PI, it must be that p_1 is a PI of h_1 [1].

From the above lemma, it can be stated that the OR of MSOPs for $h_1(X_1)$ and $h_2(X_2)$ is an SOP that represents $h_1(X_1) \vee h_2(X_2)$. Similarly, the AND of MSOPs for $h_1(X_1)$ and $h_2(X_2)$ is an SOP that represents $h_1(X_1)h_2(X_2)$. The Disjoint Computation Scheme Hypothesis also states same ideas, and they can be expressed as follows;

- $\tau(MSOP : f \vee g) = \tau(MSOP : f) + \tau(MSOP : g)$, and
- $\tau(MSOP : f \wedge g) = \tau(MSOP : f)\tau(MSOP : g)$[1].

DCSH holds in every case of $f \vee g$, but a similar statement is not true for $f \wedge g$. This is a surprising result and leads us a new expression for the DCSH specifically for $\tau(MSOP : f \wedge g)$. It is $\tau(MSOP : f \wedge g) \le \tau(MSOP : f)\tau(MSOP : g)$.

1. DCSH for f ∨ g

Use the abbreviation DCSH (∨) for direct disjunction. As was mentioned earlier, DCSH (∨), holds in every case. Specifically $\tau(MSOP : f_1 \vee f_2 \vee ... \vee f_n) =$ $\tau(MSOP : f_1) + \tau(MSOP : f_2) + ... + \tau(MSOP : f_n)$ and no f_i is a constant 1. This is not a surprising result and can be easily proved as follows.

Let $f(X)$ and $g(Y)$ be two functions, where $X \cap Y = 0$. Then $\tau(MSOP : f \vee g) = \tau(MSOP : f) + \tau(MSOP : g)$. Consider an input a, that $f(a) = 0$ then each input (a,b), where $g(b) = 1$ is covered only by the prime implicants from function *g*. Thus, it is needed $\tau(MSOP:g)$ PIs from function *g* and $\tau(MSOP:f)$ PIs from function *f*.

Consider functions $f(X)$ and $g(Y)$, where $f(x_1,x_2)=\overline{x_1}+x_2$ and $g(y_1,y_2)=y_1y_2$, the result for the direct disjunction of these two functions is $f \vee g=(\overline{x_1}+x_2)\vee y_1y_2$ and number of the PI in the MSOP representation of this new function is three according to DCSH (∨). The Karnaugh maps in the Figure 14 show this function.

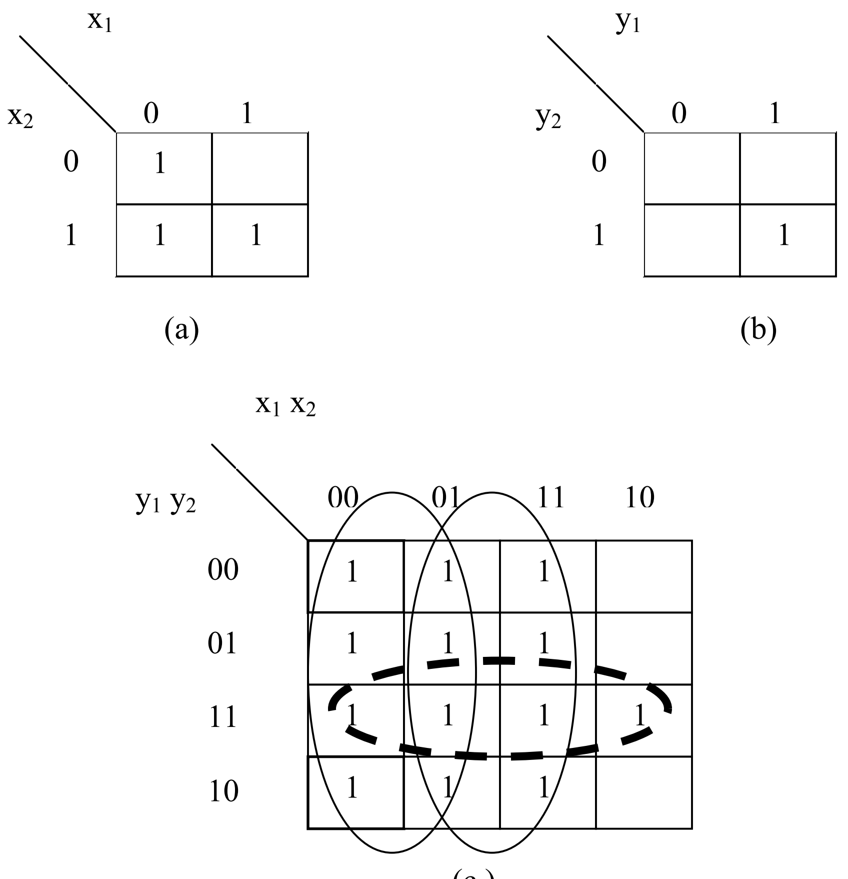


Figure 14. (a) Karnaugh Map for function f (b) Karnaugh Map for function g (c) Karnaugh Map for the new obtained function.

Choose the input *a* as 10, where $f(a)=0$, and *b* as 11, where $g(b)=1$. As can be seen from Figure 14 (c), inputs (a,b) for the new function yields 1 in the Karnaugh map representation, and this 1 comes from function *g*, which means that it can be covered only by a prime implicant that belongs to function *g*. If we pick *a* as 00, where $f(a) = 1$, and *b* as 00, where $g(b) = 0$, from Figure 14 (c) the input (a,b) for the new function is a 1 and this time it comes from the function *f*. Thus, it can be covered by a prime implicant that belongs to function *f*. Prime implicants that are marked by a thin line come from *f* and prime implicant that is marked dashed and thick line comes from *g* so that the total number of prime implicants of newly obtained function is three for its MSOP, which matches the result of the DCSH ($\vee$).

2. DCSH for *f* ^ *g*

To represent the AND DCSH, we use the abbreviation DCSH (^). Unlike DCSH ($\vee$), DCSH (^) does not hold in every case. Voight and Wegner [5] have proved this in 1989 by using a 5-variable function and in February 2001 Sasao and Butler [1] reproved it by using a 4-variable function. Also, they proved that this was the simplest counterexample to DCSH (^). Although the functions appear to be different, in reality the functions they had used were related with each other. The one that has been used by Wegner and Voight was an extended version of the one that has been used by Sasao and Butler, and this extension has been obtained by ANDing the function with one more variable. Later, it is going to be proved that these functions are related each other.

As mentioned earlier, in this chapter, the important result for us is the failure of this hypothesis rather than its success. This failure leads us to a new class of functions called orthodox functions.

As it is explained earlier DCSH (^) expresses that; $\tau(MSOP : f \wedge g) = \tau(MSOP : f)\tau(MSOP : g)$, where the literals of *f* and *g* are disjoint from each other. Although it seems reasonable to obtain the MSOP for $f \wedge g$ by simplifying *f* and *g* separately and forming a MSOP for function $f \wedge g$ by applying the law of distributivity, the result of the computation does not give us the exact MSOP for $f \wedge g$ in all cases.

To prove this, use Sasao and Butler's [1] counterexample, since it is the simplest counterexample. As shown in Figure 15, this counterexample is a 4-variable function with 6 PIs (one of which, *yw*, is not shown). Three of the six are essential PIs (denoted with dashed and thick lines) and the rest are non-essential PIs (denoted thin and solid lines). To be able to cover all the minterms of the function *f*, we need all the essential PIs and 2 of the non-essential PIs therefore, $\tau(MSOP : f) = 5$. If we consider these 5 PIs, we can represent the function f as follows; $f(x_1, x_2, x_3, x_4) = (\overline{x_1 x_2} x_3 + \overline{x_1} x_2 \overline{x_3} + x_1 \overline{x_2 x_3}) + (x_2 x_4 + x_1 x_4)$. The first pair of parenthesis encloses the essential PIs and the second encloses 2 of the 3 non-essential PIs.

Now, consider the function f^2, where $f^2 = f(X)\, f(Y)$. Function f^2 is an 8-variable function that is obtained by simply ANDing two copies of the function using two different sets of variables for multiplier. Since $f(X) = f(x_1, x_2, x_3, x_4)$ has 5 PIs, DCSH (^) suggests there are 25 PIs for the MSOP of function f^2.

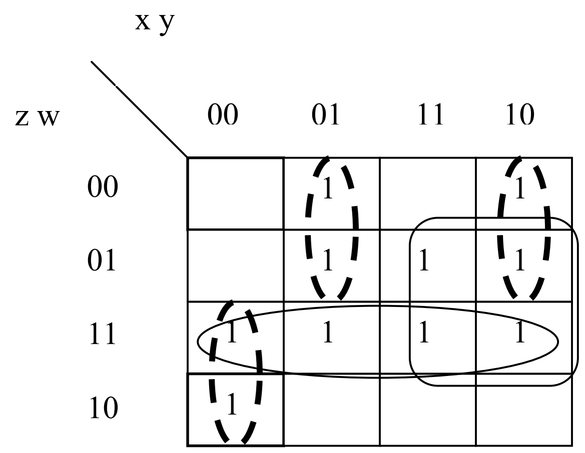


Figure 15. 4-variable counterexample.

A sum of product expression can be obtained by applying distributivity, as follows

$$f^2 = f(X)f(Y) = f(x_1, x_2, x_3, x_4) f(y_1, y_2, y_3, y_4)$$
$$= A(X,Y) \vee B(X,Y) \vee C(X,Y) \text{ [1]}$$

where, *A(X,Y)* is the product of the PIs that are essential in both function *f (X)* and *f (Y)*, *B(X, Y)* is the product of one essential and one non-essential PI, *C(X, Y)* is the product of PIs that are non-essential in both functions.

$$A(X,Y) = \bar{x}_1\bar{x}_2x_3\bar{y}_1\bar{y}_2y_3 \vee \bar{x}_1\bar{x}_2x_3\bar{y}_1y_2\bar{y}_3 \vee \bar{x}_1\bar{x}_2x_3y_1\bar{y}_2\bar{y}_3 \vee \bar{x}_1x_2\bar{x}_3\bar{y}_1\bar{y}_2y_3 \vee \bar{x}_1x_2\bar{x}_3\bar{y}_1y_2\bar{y}_3$$
$$\vee \bar{x}_1x_2\bar{x}_3y_1\bar{y}_2\bar{y}_3 \vee x_1\bar{x}_2\bar{x}_3\bar{y}_1\bar{y}_2y_3 \vee x_1\bar{x}_2\bar{x}_3\bar{y}_1y_2\bar{y}_3 \vee x_1\bar{x}_2\bar{x}_3y_1\bar{y}_2\bar{y}_3$$
$$B(X,Y) = \bar{x}_1\bar{x}_2x_3y_2y_4 \vee \bar{x}_1\bar{x}_2x_3y_1y_4 \vee \bar{x}_1x_2\bar{x}_3y_2y_4 \vee \bar{x}_1x_2\bar{x}_3y_1y_4 \vee x_1\bar{x}_2\bar{x}_3y_2y_4 \vee x_1\bar{x}_2\bar{x}_3y_1y_4$$
$$\vee x_2x_4\bar{y}_1\bar{y}_2y_3 \vee x_2x_4\bar{y}_1y_2\bar{y}_3 \vee x_2x_4y_1\bar{y}_2\bar{y}_3 \vee x_1x_4\bar{y}_1\bar{y}_2y_3 \vee x_1x_4\bar{y}_1y_2\bar{y}_3 \vee x_1x_4y_1\bar{y}_2\bar{y}_3$$
$$C(X,Y) = x_2x_4y_2y_4 \vee x_2x_4y_1y_4 \vee x_1x_4y_2y_4 \vee x_1x_4y_1y_4 \text{ [1].}$$

It is obvious that, *A (X, Y)* $\vee$ *B (X, Y)* $\vee$ *C (X, Y)* gives a total of 25 PIs as expected. However, f^2 can be represented using only 24 PIs instead of 25, where *C (X,Y)* is replaced by $x_3x_4y_3y_4 \vee x_2x_4y_2y_4 \vee x_1x_4y_1y_4$ and *A (X, Y)*, *B (X, Y)* remain same. This new SOP for f^2 can be verified to be an MSOP by Espresso. This is a counterexample to DHCP. It shows that decomposing a function into subfunctions on disjoint sets of variables (AND disjoint bi-decomposition), minimizing the two SOP's separately and applying the law of distributivity does not always yield an MSOP.

This counterexample's result leads us a new class of functions, which always yields the MSOP when we apply DHCP (^), called *orthodox functions*. Recent researches that done by Sasao and Butler [1] show all symmetric functions, functions with three or fewer variables, all unate functions, many benchmark functions and few random functions are included in this new class.

A function $f(X)$ is an *orthodox function*, if and only if the number of PIs in the MSOP representation of the function $f(X)$ is equal to the number of minterms in its maximal independent set. Algebraically this can be expressed as; $\tau(MSOP : f) = \eta(f)$.

B. SAMPLE ORTHODOX FUNCTIONS

Now, different classes of orthodox functions are demonstrated.

- The orthodox function *f*, where $f(x_1, x_2, x_3) = x_1\bar{x}_2 \vee x_2\bar{x}_3 \vee x_3\bar{x}_1$ belongs to the three or fewer variables subclass. This function is orthodox, since $\tau(MSOP : f) = \eta(f) = 3$. In Figure 16, the minterms that belong to the maximum independent set of minterms are marked by ✧, and circled minterms are the PIs of the MSOP. Also, for this example, it is interesting that there is another maximum independent set of minterms and another MSOP.

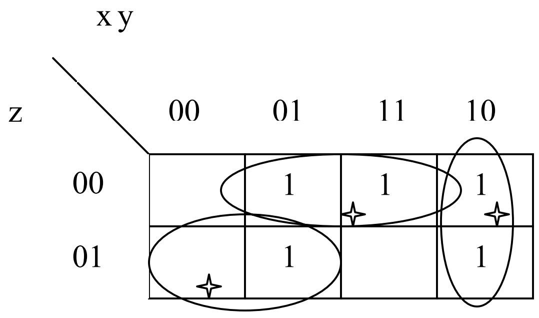


Figure 16. An orthodox function that belongs to the three or fewer variables subclass.

- The next orthodox function *f*, where $f(x_1, x_2, x_3, x_4) = x_1x_2 \vee x_3x_4$ belongs to the unate functions subclass. This function is an orthodox function, since $\tau(MSOP : f) = \eta(f) = 2$. In Figure 17 the minterms marked by ✧ are the ones that belong to the maximum independent set of minterms, and circled minterms are the PIs that belong to the MSOP.

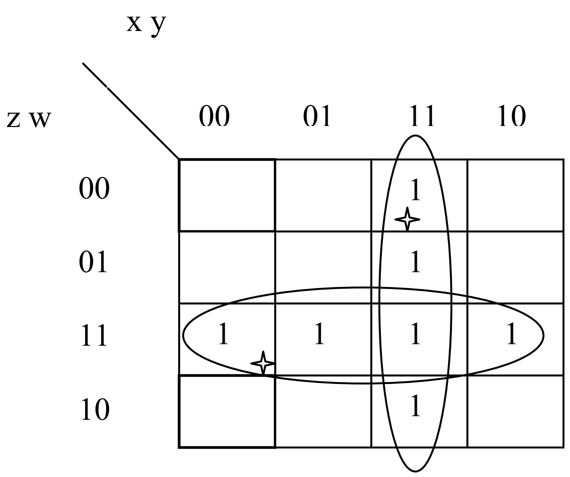


Figure 17. An orthodox function that belongs to the unate function subclass.

- Third and the last example orthodox function *f*, where $f(x_1,x_2,x_3,x_4) = x_1\overline{x_2 x_3 x_4} + \overline{x_1}x_2 x_3 x_4 + x_1 x_2\overline{x_3 x_4} + \overline{x_1 x_2}x_3\overline{x_4} + \overline{x_1 x_2 x_3}x_4 + x_1x_2x_3\overline{x_4}$ $x_1x_2\overline{x_3}x_4 + x_1\overline{x_2}x_3x_4 + \overline{x_1}x_2x_3x_4$
  belongs to the symmetric functions subclass. This function is an orthodox function, since $\tau(MSOP : f) = \eta(f) = 8$. In Figure 18 the minterms marked by ✧ are the ones that belong to the maximum independent set of minterms, and circled minterms are the PIs that belong to the MSOP.

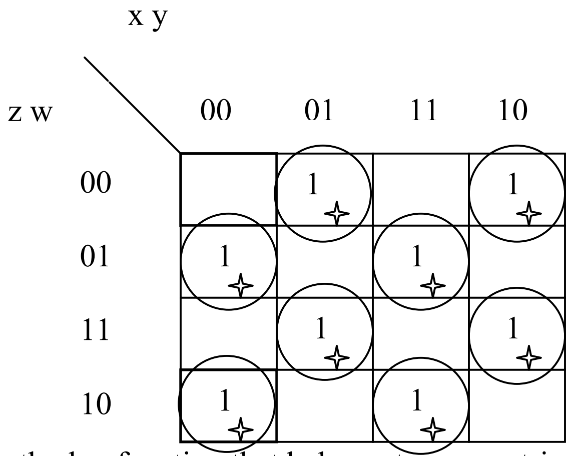


Figure 18. An orthodox function that belongs to symmetric function subclass.

C. THEOREMS AND OBSERVATIONS

Orthodox functions are new concept. Therefore, there are not so many theorems related with this type of function. The known theorems are the ones that had been derived by Sasao and Butler [1], and these theorems focus more on the determination of the orthodox functions than on their specific properties.

In the following paragraphs, the known theorems of the orthodox functions will be discussed and, in the experimental results chapter, new observations and conjectures related with this new class of functions will introduced.

1. Theorem 1

Let functions f (X) and g (Y) be orthodox functions. If variable subsets X and Y are disjoint, then the function f (X) g (Y), is also an orthodox function [1].

Consider the function *f*, where $f(x_1, x_2) = \overline{x_1}x_2 + x_1\overline{x_2}$ and function g, where $g(y_1, y_2) = y_1$. The function *z*, which is obtained by the logical AND of functions *f* and *g*, can be represented as $z(x_1, x_2, y_1, y_2) = \overline{x_1}x_2y_1 + x_1\overline{x_2}y_1$ and this function also is an orthodox function, since $\tau(MSOP : f) = \eta(f) = 2$. Figure 19 shows the Karnaugh map representations of the functions.

2. Theorem 2

Let functions f (X) and g (X) be NP-equivalent. f is an orthodox function if and only if g is orthodox [1].

The proof of this theorem is straightforward. Since the NP-equivalent functions are obtained from the same subset of variables' permutation and complementation, we can easily declare that the MSOP of $f(X)$ can be formed from $g(X)$, by a suitable complementation and permutation of variables that belong to function $g(X)$. The converse of the case is also true. Therefore, if $g(X)$ has α independent minterms, so does function $f(X)$. Similarly the MSOP of the *f* and *g* have the same number of PI's.

Consider the functions *f* and *g*, where $f(X) = x_1 + \overline{x_2}$, $g(X) = \overline{x_1} + x_2$. Function *g* is obtained by permutation of the variables of function f. Since both representations are the MSOP representations of the functions, we do not need any further simplifications. Now, if we consider the function *f*, we can observe, it has 2 independent minterms. So does function *g* from the previous paragraph's discussion; they both have 2 PIs in their MSOP representation, and so $\tau(MSOP : f) = \eta(f) = \tau(MSOP : g) = \eta(f) = 2$.

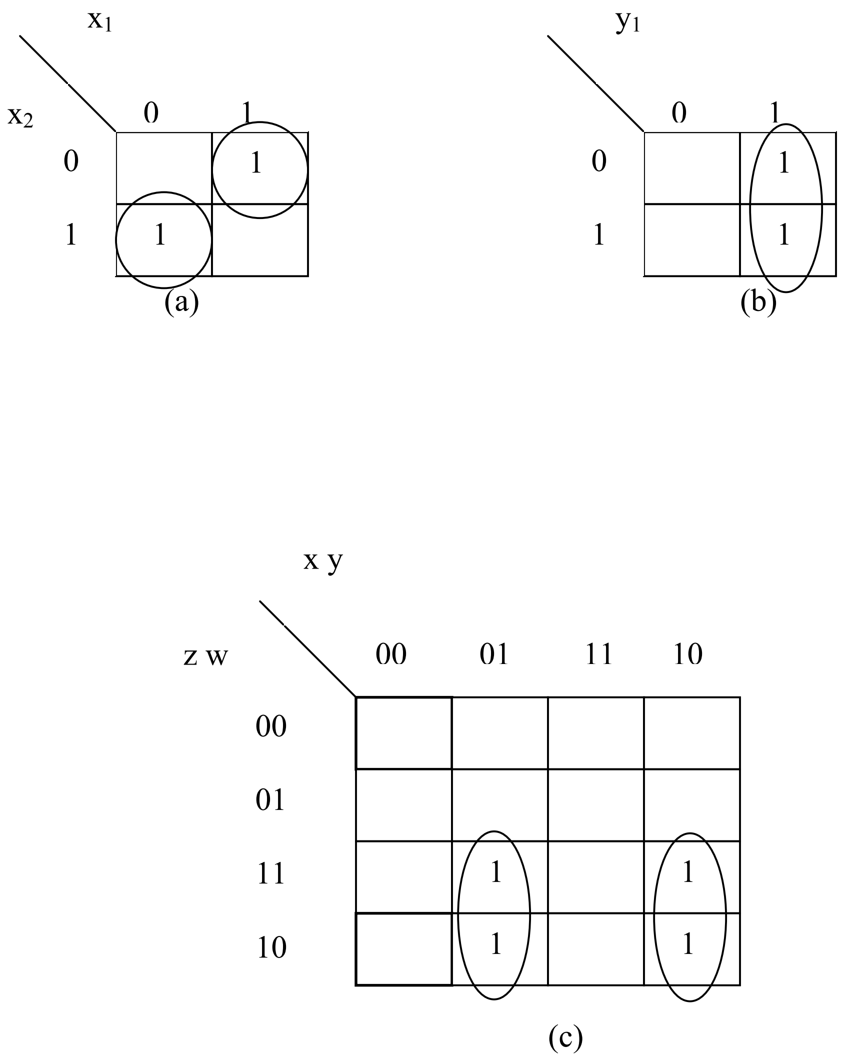


Figure 19. Karnaugh map representations.

3. Theorem 3

If f's MSOP representation consists of only essential PIs then f is an orthodox function [1].

To prove this theorem, it is sufficient to consider distinguished minterms. This concept suggests that, each essential PI covers a minterm that is covered by only that PI, which means that in the maximum independent set of minterms, we will have at least the number of essential PIs many independent minterms so if we have only essential PIs for a function this function is an orthodox function since, $\tau(MSOP : f) = \eta(f)$.

Consider f, where function $f(x_1, x_2, x_3, x_4) = \overline{x_1} x_2 \overline{x_3} + x_1 \overline{x_2}\,\overline{x_3} + \overline{x_1}\,\overline{x_2} x_3$. This function consists of only essential PIs, as can be seen from its Karnaugh map representation in Figure 20. So, each essential PI covers a distinguished minterm, and we have 3 distinguished minterms and 3 essential PIs, which perfectly matches the definition of orthodox functions.

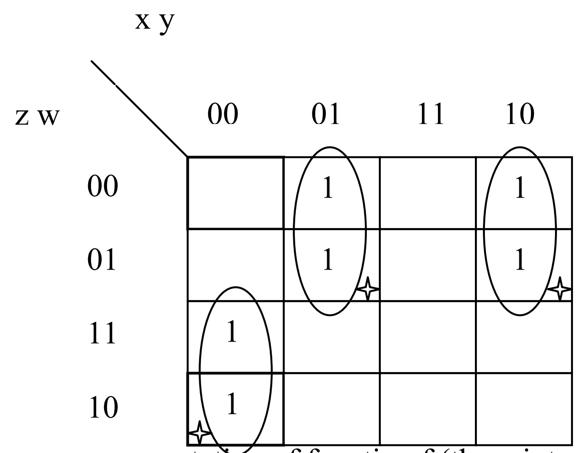


Figure 20. Karnaugh map representation of function f (the minterms that denoted by "✧" are the distinguished minterms).

D. CONCLUSIONS OF THE CHAPTER

Orthodox functions, their known properties, and disjoint computation scheme hypothesis for logical AND and OR operations addressed in this chapter. After the introduction of the orthodox functions three example orthodox functions were demonstrated in this chapter also. The non-orthodox function concept addressed in Chapter IV of this thesis, same chapter also proposes an algorithm, Algorithm 3, to create non-orthodox functions.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. NON-ORTHODOX FUNCTIONS

It has been shown experimentally that the fraction of n-variable functions that are orthodox approaches 0 as n approaches infinity [1]. Figure 21 depicts the orthodox and non-orthodox functions as disjoint subsets of all functions.

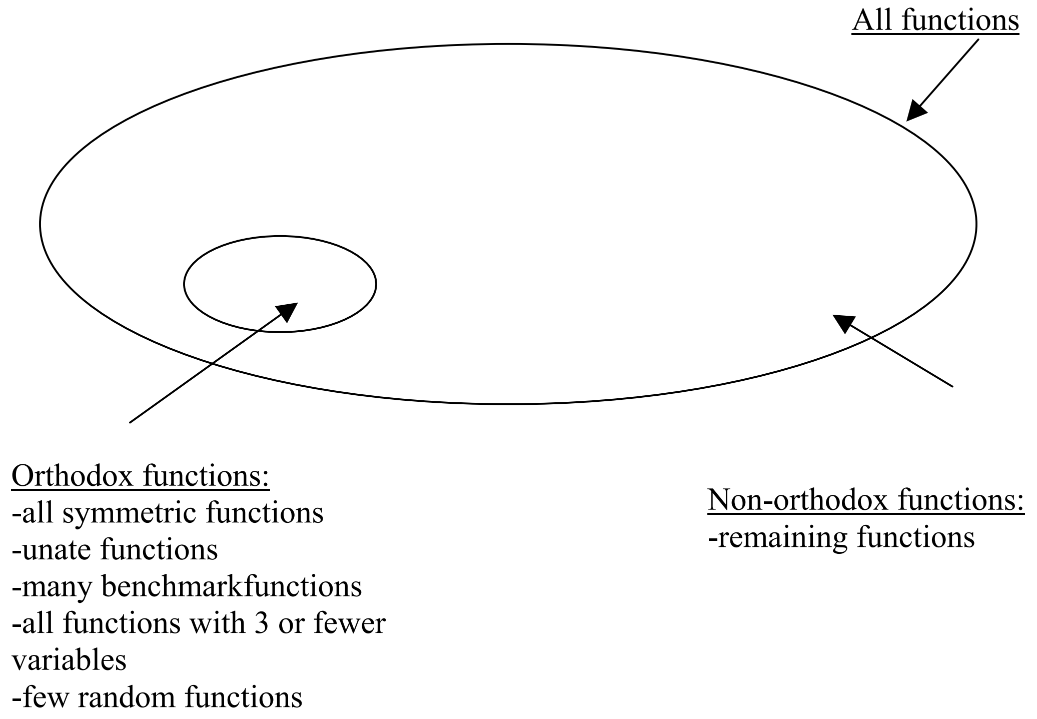


Figure 21. Set of the all functions divided among orthodox and non-orthodox functions.

People may think that the number of orthodox functions is greater than the number of non-orthodox functions because, among the functions that people can manipulate without help of computer, the number of the functions that are orthodox is greater than the number that are non-orthodox. This is true of 4-variable functions, 5-variable functions and 6-variable functions; the percentage of the non-orthodox functions is small in these types of functions. Table 5 can give us a quick idea about the percentage of the non-orthodox functions that have 4 to 10-variables [1].

Number of Variables	Percentage of Non-orthodox functions (%)
4	0.4
5	1
6	4
7	13
8	34
9	81
10	100

Table 5. Percentage of the Non-orthodox functions within 4 to 10-variable functions

After this introduction, non-orthodox functions discussion may proceed with the research results of Sasao and Butler [1].

A. FOUR-VARIABLE NON-ORTHODOX FUNCTIONS

There are 65536 functions of 4-variables, 2^{16}, which can be divided into 402 NP-equivalent sets. It has been verified that only 4 NP-equivalence sets are non-orthodox. A representative from each NP- equivalence class can be obtained by substituting 1s and 0s for the don't cares in Figure 22. Each representative function is NP-equivalent to 63 other functions. As a result, we have totally 64*4 = 256 non-orthodox functions in 4-variable functions, which gives us the percentage of 0.4% shown in Table 5[1].

z w \ x y	00	01	11	10
00	-	1		1
01		1	1	1
11	1	1	1	1
10	1		-	

Figure 22. Four variables non-orthodox functions.

1. Properties of the 4-Variable Non-orthodox Functions

As explained earlier, there are 4 NP-equivalent classes among the 4-variable functions and representatives of each class can be obtained by substituting 1s and 0s in the don't cares in the Figure 22.

Therefore, one class representative has two zeros in minterms $x_1x_2x_3x_4 = 0000$ and 1110 and can be denoted as NP_{00}. Another class representative has a 1 in minterm $x_1x_2x_3x_4 = 0000$, a 0 in minterm $x_1x_2x_3x_4 = 1110$ and can be denoted as NP_{01}. Still another class representative has a 0 in minterm $x_1x_2x_3x_4 = 0000$, a 1 in minterm $x_1x_2x_3x_4 = 1110$ and can be denoted as NP_{10}. The last class representative has two 1s in minterms $x_1x_2x_3x_4 = 0000$ and 1110 and can be denoted as NP_{11}.

It has been verified by Sasao and Butler [1] that when 4-variable non-orthodox functions are squared by using disjoint variable sets they provide less PIs in the MSOP of resultant function then expected. So 4-variable non-orthodox functions have the property that $\tau(MSOP : f^2) < \tau(MSOP : f)^2$.

The experiments with the NP-equivalence sets NP_{00}, NP_{01}, NP_{10} and NP_{11} have verified that when the members of these classes logically ANDed with each other by using disjoint variable sets the resultant functions yield less PIs than expected from the DCSH ($\wedge$). So, it can be stated that 4-variable non-orthodox functions also have the property that $\tau(MSOP : f \wedge g) < \tau(MSOP : f)\tau(MSOP : g)$, where $f, g \in \{NP_{00}, NP_{01}, NP_{10}, NP_{11}\}$ and the difference is always 1.

B. CREATING A NON-ORTHODOX FUNCTION

Of the five variables functions, it is estimated that 1% of the total are non-orthodox. The only known 5-variable non-orthodox function is the one proposed by Voight and Wegner [5] (and the others that can be derived from it by a permutation and complementation of the variables of the proposed one). This function is closely related to the known examples of the 4-variable non-orthodox functions. This is going to be demonstrated later in this thesis.

But, it is known from experiments that non-orthodox functions represent the vast majority of functions with nine or more variables. Thus, there is a gap in our understanding of these functions. It is the goal of this section to fill this gap.

Below is a procedure by which one can create a non-orthodox function with desired number of variables. To understand this procedure, consider the following.

Consider the 4-variable non-orthodox functions representative, Figure 23 is going to be its Karnaugh map representation. It can be observed that the non-essential PIs of the function f give the non-orthodox property to the function. To understand how this happens, consider the following.

1. Discussion for Non-orthodox Functions

It is explained that the non-orthodox function property is achieved by the help of the non-essential PIs. Consider the definition of the non-orthodox function, the only requirement is, a difference between the number of the prime implicants of the sum of products representation of the function and the number of the elements of the maximal independent minterm set. Also, if one considers the definitions of the non-essential prime implicants and independent minterms set, it is obvious that the required difference can only be obtained from the non-essential prime implicants since every essential prime implicant has a minterm that is counted in $\eta(f)$.

In the case of non-essential prime implicants, each minterm of the prime implicant can be covered by more than one non-essential or essential prime implicant. So, under certain conditions, it is impossible to find an independent minterm from these prime implicants, e.g. in Figure 23, there are 3 non-essential PIs in the Karnaugh map representation of the function, and only 1 minterm among these 3 PIs' minterms is counted in $\eta(f)$.

Let's continue the examination of the 4-variable non-orthodox function's Karnaugh map representation. Consider the non-essential prime implicants. If the minterms associated with them are isolated from the whole function, the Karnaugh map shown in Figure 24 is going to be obtained, which shows the middle two rows of the Karnaugh map in Figure 23.

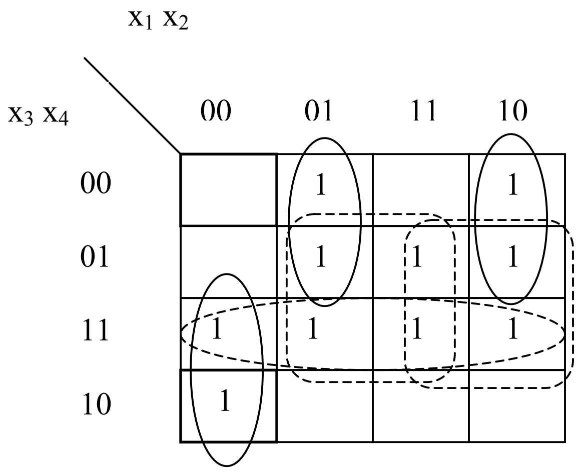


Figure 23. 4-variable non-orthodox function. Dashed lines show the non-essential PIs, solid lines show essential PIs.

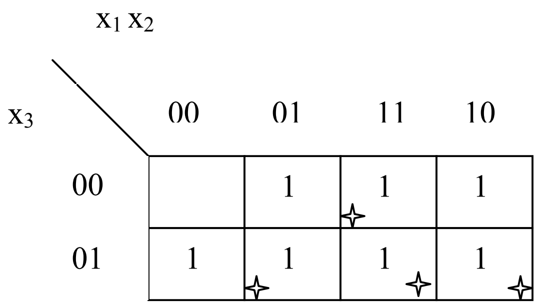


Figure 24. Middle two rows of Figure 21.

Therefore, eliminating the minterms in Figure 23 that are covered by only essential PIs, yields the minterms in Figure 24. The minterms in Figure 24 marked with “✧“ form a *majority function* and the non-essential PIs that are created by the minterms of this majority function determine the difference between the number of the PIs in the MSOP of *f*, $\tau(MSOP : f)$ and the number of the minterms in the maximal independent minterms set of *f*, $\eta(f)$, depending on the number of the PIs that needed in the MSOP of *f* to cover all the minterms. Also, note that isolating the minterms that are associated only with essential PIs of the 4-variable non-orthodox function yields a new function with 3-

variables, where this 3-variable function includes the majority function that is formed by the marked minterms of the Figure 24.

2. Steps of the Algorithm 3

After this discussion, one can consider that there is a common point in the Non-Orthodox function's Karnaugh Map representation, and it can be used to produce new Non-Orthodox functions with more variables. Consider a method for producing non-Orthodox functions by using this idea:

Algorithm 3:

1. Determining the number of variables of the non-orthodox function, *n*, that is going to be constructed, where n is even.

Subtract 1 from *n*, to determine the number of variables of the function that holds the majority function as explained in the previous section.

2. Determine the *weak minterms* of the function (function that we determine the number of the variables in the previous step), to find the minterms of the essential PIs of the newly generated non-orthodox function. These are minterms that are covered by at least one essential and at least one non-essential PI. For example, consider Figures 23 and 24, eliminating the minterms that are covered only by essential PIs accomplished the subtracting 1 from the variable number of the non-orthodox function step. Thus, we obtained the function that holds the majority function (discussed in the previous part). So that, the minterms that unmarked in Figure 24 are the weak minterms.

a. To create minterms that belong to the essential PIs of the non-orthodox function that wanted to be constructed, append a dash to the end of each weak minterm. Then, expand it by substituting 0 and 1 for dash; i.e. $x_1x_2x_3 = 001$ is a weak minterm in Figure 24, appending a dash yields 001-, expanding it yields 0010 and 0011, 4-variable minterms of the function *f* shown in Figure 23.

3. Finally, to obtain the minterms that are covered by the non-essential PIs of the non-orthodox function that wanted to be constructed, the strong minterms should be determined. And, opposite of the weak minterms appending a 1 at the

end of these minterms is sufficient to expand them. For example, consider the minterm 110 in Figure 24. It is a strong minterm, adding a 1 to the end of it yields 1101. This minterm is covered by one of the non-essential PI of non-orthodox function, as shown in Figure 23.

Consider how Algorithm 3 can be used to create a 6-variable non-orthodox function.

- Subtract 1 from the number of variables in the non-orthodox function to determine the number of the variables of the function that is going to hold the majority function.

To obtain the minterms covered by the essential PIs of the non-orthodox function, that is wanted to be constructed by using this algorithm;

- Determine the weak minterms of the 5-variable function, in running example, a 5-variable function is considered, so no weak minterm can have more than 2 1s in it. Table 6 is a suitable way to represent the weak minterms.
- In Table 6, each row corresponds to a set of weak minterms. For example, consider the first row of Table 6. Substitute 1s and 0s for dashes such that there are no more than 2 1s. This yields the weak minterms 10000, 11000, 10100, 10010 and 10001.

x_1	x_2	x_3	x_4	x_5
1	-	-	-	-
0	1	-	-	-
0	0	1	-	-
0	0	0	1	-
0	0	0	0	1

Table 6. The weak minterms.

- Next, append dashes to each weak minterm and expand it. For example, consider the weak minterms that obtained in previous step, adding dashes to the end of their cube notations and substituting 1s and 0s for these dashes yields following 6-variable minterms; 100000-100001, 110000-110001, 101000-101001, 100100-100101, 100010-100011. To find all the minterms that covered by the essential PIs of the 6-variable non-orthodox function that wanted to be created, apply the same steps that were applied to each row of Table 6. Table 8 shows all the minterms that covered by essential PIs of the 6-varable non-orthodox function.

To obtain the minterms that are covered by the non-essential PIs of the non-orthodox function that wanted to be generated;

- Determine the strong minterms of the 5-variable function. Table 7 shows the strong minterms of the 5-variable function.
- On the contrary to weak minterms, append only 1 to the cube notations of the strong minterms. This yields the 6-variable minterms that covered by the non-essential PIs of the non-orthodox function we want to generate. The first 5 columns of the Table 8 show the 5-variable strong minterms, Table 8 shows the the minterms that covered by non-essential PIs of the non-orthodox function.

x_1	x_2	x_3	x_4	x_5	x_6
0	1	0	1	1	1
0	1	1	1	0	1
0	1	1	1	1	1
0	0	1	1	1	1
0	1	1	0	1	1
1	1	0	0	1	1
1	1	1	0	0	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	1	0	1
1	1	1	1	1	1
1	1	1	0	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	0	1	1	0	1
1	0	1	0	1	1

Table 7. The strong minterms.

After applying the above discussion to the weak minterms of the 5-variable function, the minterms shown in Table 8 were obtained as the minterms of the essential PIs of the 6-variable non-orthodox function.

# of the row of Table 6	x_1	x_2	x_3	x_4	x_5	x_6
From row 1	1	0	0	0	0	0/1
From row 1	1	1	0	0	0	0/1
From row 1	1	0	1	0	0	0/1
From row 1	1	0	0	1	0	0/1
From row 1	1	0	0	0	1	0/1
From row 2	0	1	0	0	0	0/1
From row 2	0	1	1	0	0	0/1
From row 2	0	1	0	1	0	0/1
From row 2	0	1	0	0	1	0/1
From row 3	0	0	1	0	0	0/1
From row 3	0	0	1	1	0	0/1
From row 3	0	0	1	0	1	0/1
From row 4	0	0	0	1	0	0/1
From row 4	0	0	0	1	1	0/1
From row 5	0	0	0	0	1	0/1

Table 8. All the minterms that belong to the essential PIs of the 6-variable non-orthodox function that is generated by Algorithm 1.

As it is seen from Tables 7 and 8 there are a total of 46 minterms of the non-orthodox function.

3. Verifying the Non-orthodox Function Property

After specifying all minterms of the function, it is necessary to determine whether the created function has the non-orthodox property or not. To make this decision, it is needed to be known the number of the PIs in generated function's minimum sum of products representation and number of the minterms in its maximum independent set of minterms. Place all the minterms that we determined into a Karnaugh map to be able to find the number of the minterms in the independent minterms set, $\eta(f)$ and $\tau(MSOP : f)$.

Figure 25 shows the Karnaugh map representation of function produced by Algorithm 3. Minimizing this function yields a total 13 PIs; 10 essential and 3 non-essential. Espresso was used for the minimization. During the minimization –Dexact flag was used to obtain an exact expression for MSOP. Therefore $\tau(MSOP : f) = 13$.

To determine the independent minterms, consider the essential PIs and the non-essential PIs. For each essential PI, there is 1 independent minterm, and overall essential PIs is 10. If we consider the non-essential PIs, there is only 1 independent minterm. Overall, there are 11 independent minterms and so $\eta(f) = 11$. This is shown by a “x” in Figure 25, and all the independent minterms that come from essential PIs are marked also by “✧”.

Since $\tau(MSOP : f) \neq \eta(f)$, it can be stated that the function derived from the Algorithm 3 is a non-orthodox function.

C. CONCLUSIONS OF THE CHAPTER

In this chapter of this thesis, the non-orthodox function concept was introduced to the reader. To do this the known simplest non-orthodox functions, 4-variable non-orthodox functions were used. Also, an algorithm, Algorithm 3, was proposed in this chapter. This algorithm constructs non-orthodox functions with 2n-variables, where n is even. In the “Creating a non-orthodox function” part of the thesis, Algorithm 3 is completely discussed and correctness of the results of this algorithm is also verified by evaluating the function that is created by Algorithm 3 in respect to official non-orthodox function definition. Chapter V, the following chapter, is the part of this thesis that presents the results of the experimental research done with non-orthodox and orthodox functions.

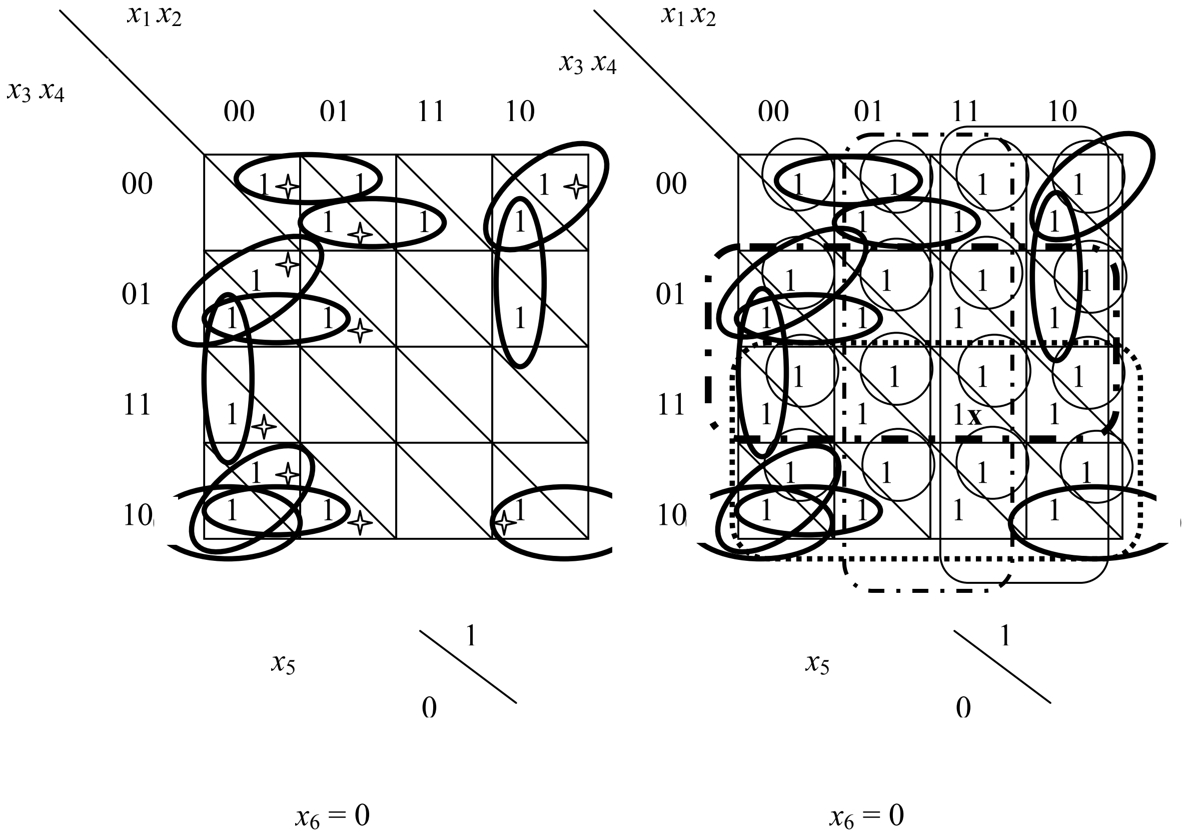


Figure 25. 6-variable non-orthodox function.

V. EXPERIMENTAL RESULTS

Experiments were conducted using Espresso and 6 java programs. The goal of these experiments is to increase our knowledge of orthodox and non-orthodox functions.

Because it is a basic tool of experiments, Espresso has been discussed in detail in the introduction. In the following sections, brief explanations of the other software, all of which are java programs, are given. These are mostly used to create input files for Espresso (except the YaratNon.java). YaratNon.java, on the other hand, is used to create the representative function of the non-orthodox functions with a given number of variables. It uses the algorithm discussed in Chapter 3.

A. YARATNON.JAVA

This program is used to obtain the representatives of families of non-orthodox functions with the given number of variables. The number of variables must be even number due to the algorithm of the program. This program has two inputs; the name of the output file, which has the extension ".es", and number of the variables of the majority function, which is an odd number. This program implements the Algorithm 1.

The java source code of the YaratNon.java is shown in Appendix A.

B. FAMILY.JAVA

This program creates a family of functions from one representative function produced by YaratNon.java. It determines the minterms that can be treated as don't cares. To accomplish this task, the program obtains the minterms from the user that are potentially don't cares. Then, it creates new functions by inserting the potential don't cares into the original non-orthodox function. Each newly created function is written to a new file that has an extension ".es". Family.java uses the On-set representation while writing the functions into different files, because these files are used as input files for Espresso and Espresso uses the On-set representation as default.

Inputs to the program include; the name of the original non-orthodox function's file, the file created by YaratNon.java, each potential don't care minterm that the user wants to investigate and the output file name containing the newly created functions.

For example, if you have a representative function for 6-variable non-orthodox functions, and you want to specify 2 minterms as don't cares, Family.java will provide you 4 new functions in 4 different files. Each file has the same name with a different number appended; i.e let the name of the output file entered be "experience.es". Then "experience0.es" will be given to the first function, "experience1.es" the second, up to 3.

The java source code of the Family.java is shown in Appendix B.

C. SONKARAR.JAVA

Family.java's task is to create a number of input files for Espresso, depending on the number of minterms specified as don't cares. Espresso creates an output file with an extension ".out" for each function that has been minimized by it and SonKarar.java's task is to read each of these output files and determine whether the function is an orthodox function or non-orthodox function. To make this decision, SonKarar.java uses simple logic, described as follows.

- SonKarar.java goes thorough the Espresso output file of each function, and checks the PIs. SonKarar.java controls the cube notations of each PI , increments its counter whenever it encounters a PI that has a 1 at the very last literal of its cube notation. After SonKarar.java done either one of the two cases holds:
- Case 1: Function's MSOP may include more that 1 PI such that their cube notations have a 1 for the last literal, then the function that is under test is a *non-orthodox function*.
- Case 2: Function's MSOP may include exactly 1 PI such that its cube notation has a 1 for the last literal, then the function under test is an *orthodox function*.

Although it looks like a simple logic, the algorithm discussed above should be proved since, decisions taken by SonKarar.java directly affects the experimental results of this chapter. Following is the proof of the above algorithm;

It is known from Chapter III that Algorithm 3 is used to create the basic n-variable non-orthodox function, where n is even. Let the function created by Algorithm 3 be f, where f can be written as follows;

$$f = \left(S_{1,2,\ldots,\frac{n}{2}-1} \right) \overline{x_n} \vee (x_1 \vee x_2 \vee \ldots \vee x_{n-1}) x_n .$$

Let MI be a maximal independent set of f_1. Among the minterms in the subfunction f_1 of f, where f_1 is;

$f_1 = S_{\frac{n}{2}+1,\frac{n}{2}+2,\ldots,\frac{n}{2}+\frac{n}{2}} (x_1, x_2, \ldots, x_{n-1}, 1)$, where n is the number of the variables of f.

MI may only have 1 minterm.

f_1 consists of minterms that all belong to the non-essential PIs of function f. These minterms can be covered by 5 non-essential PIs as shown in Figure 25. Table 9 shows the cube notations of these non-essential PIs, and also observe from the same table that all PIs of function f_1 has a 1 in their last literal of cube notations.

x_1	x_2	x_3	x_4	x_5	x_6
1	-	-	-	-	1
-	1	-	-	-	1
-	-	1	-	-	1
-	-	-	1	-	1
-	-	-	-	1	1

Table 9. Cube notations of the non-essential PIs of f.

Choose two independent minterms among these 5 PIs, e.g. 001111 and 110101. From Table 9, it can be observed that the chosen minterms are both covered by 4th non-essential PI. If the minterms belong to f_1 are checked, it can be seen that there is no minterm that is covered by only 1 PI. Thus, no more than 1 minterm can be picked as an independent minterm among these 16 minterms.

It follows that, if the function under test has more than 1 PI that has a 1 at the last literal of its cube notation, then the number of the minterms in its independent minterms set is less than the number of the PIs in its MSOP. Thus, this function is a non-orthodox function.

Also, the same computer program can be used to determine the types of functions created by Family.java because, Family.java creates functions by substituting 1s and 0s to the minterms provided by the user. The user cannot pick minterms from the right Karnaugh map of Figure 25, since there is only one minterm that is not assigned a 1 in this Karnaugh map, and to assign a 1 to this minterm causes to f_1 to be 1. And, this makes the n-variable function f a $(n\text{-}1)$-variable function that consists of only essential PIs. As mentioned in Chapter III, this type of functions is always orthodox.

Thus, in the usage of Family.java, the user has to provide minterm/minterms to the program from the left Karnaugh map in Figure 25. It shows the essential PIs of the function f, and the insertion of new minterms in this Karnaugh map will not affect the PIs (circles) of left Karnaugh map in Figure 25. That is, during the minimization of the functions, the biggest circles should be picked to cover the minterms to be able to remove as many literals as possible from the PIs of the MSOPs, and the biggest 5 circles (PIs) are listed in Table 9. No matter how many uncovered minterms remained in the right Karnaugh map of Figure 25, at least one of these circles we should be picked. It follows that the same idea used in YaratNon.java can be used, to determine the types of the functions that are created by Family.java.

The java source code of the SonKarar.java is shown in Appendix C.

D. CARPIMTABLOSU.JAVA

This program's task is to square a given non-orthodox function. To perform the square operation, Carpimtablosu.java logically ANDs 2 copies of the given function by using 2 disjoint variable sets for each copy. It gets the file name of the original function that we wish to square and an output file name for the squared functions from the user. (CarpimTablosu.java uses the given names as bases and appends a counter to these bases to point the proper function.)

Let the base name for the total 16 non-orthodox functions be "birol", and let the base name for the squared version of these functions be "filiz". Then, the program reads the files birol0.es, birol1.es and so forth up to birol15.es, squares each of the read file and names them filiz0.es, filiz1.es respectively.

The java source code of the CarpimTablosu.java is shown in Appendix D.

E. COMPARE.JAVA

Compare.java attempts to answer the question "*Are there any AND bi-decomposable functions in which the application of the law of distributivity to the MSOPs of component functions produces an SOP with many more PIs than in the MSOP?*" [1].

To do this, Compare.java goes through Espresso's output files and gets the number of the PIs in the MSOPs of the functions that are created by CarpimTablosu.java and Family.java. Compare.java squares the PIs that belong to the MSOP of the functions created by Family.java, to obtain the number of the PIs in the MSOP of f^2 when the law of distributivity is applied. Then, Compare.java subtracts the resultant number from the PIs number taken from the Espresso output files for the functions created by CarpimTablosu.java. Compare.java writes the results of the subtractions to an output file.

The java source code of the Compare.java is in Appendix E.

F. ESPRESSO2.JAVA

Espresso2.java is used to compute the resultant function of the logical ANDing operation between 2 any n-variable function.

Espresso2.java gets the input functions as On-sets from the user and also provides the resultant function with On-set representation. Results of this program can used as input files for Espresso. Espresso2.java also gets the name of the output file to write the resultant function.

The java source code of the Espresso2.java is shown in Appendix F.

G. USAGE OF THE JAVA PROGRAMS AND ESPRESSO IN EXPERIMENTS

To conduct experiments with orthodox/non-orthodox functions, programs explained above must be used in a special order, because of the fact that the result files of a program are going to be the input files of another program.

Figure 26 and 27 depicts the usages of the programs for different kinds of experiments. To be able to determine the representatives of the non-orthodox functions' NP-equivalence sets, the programs must be used in the order that is depicted by Figure 26. Following is a brief explanation of the figure;

- Create the base n-variable non-orthodox function using YaratNon.java.
- The output file of the YaratNon.java becomes the input of the Family.java. Family.java substitutes 1s and 0s for the minterms that are provided by user. Then, it inserts the newly created minterms to the minterms set of the n-variable base non-orthodox function to create new functions.
- Espresso minimizes all the functions that are created by Family.java. Espresso writes the results into different output files for each function.
- Output files of Espresso become the input files for SonKarar.java. SonKarar.java goes thorough the output files to determine whether there is an orthodox function in the functions that are created by Family.java or not. If there is no orthodox function, then the provided set of minterms can be treated as don't cares. Otherwise, the last provided minterm are excluded from the set.

- The user continues to provide the minterms until there is no minterm to be checked.

Figure 27 depicts the usage of the programs to determine the penalty of minimization a bi-decomposable function by applying law of distributivity and minimizing without decomposing the function into subfunctions. Following is a brief explanation of the flow chart that is depicted in the Figure 27.

- Result files (each one represents a function) of Family.java become the inputs of the CarpimTablosu.java, when it is decided that they are non-orthodox functions.
- CarpimTablosu.java squares each function and writes the results to different files for each function. These files become the input files for Espresso.
- Espresso minimizes all the squared functions.
- Compare.java finds the difference between $\tau\left(MSOP : f^2\right)$ and $\tau(MSOP)^2$ by using the output files of Espresso, as explained in part E.

To be able to verify the correctness of the Java programs, they were used to conduct experiments with all 4-variable non-orthodox functions and the results were compared with the results generated by Sasao and Butler's computer program [1]. This comparison showed that Java programs that developed for the experimental research of this thesis worked correctly. Then, exploration of 6-variable non-orthodox functions started. During the experiments, 924,288 6-variable functions have been created. The experiments were conducted over a period of 4 weeks. The analysis of the generated data took 5 weeks.

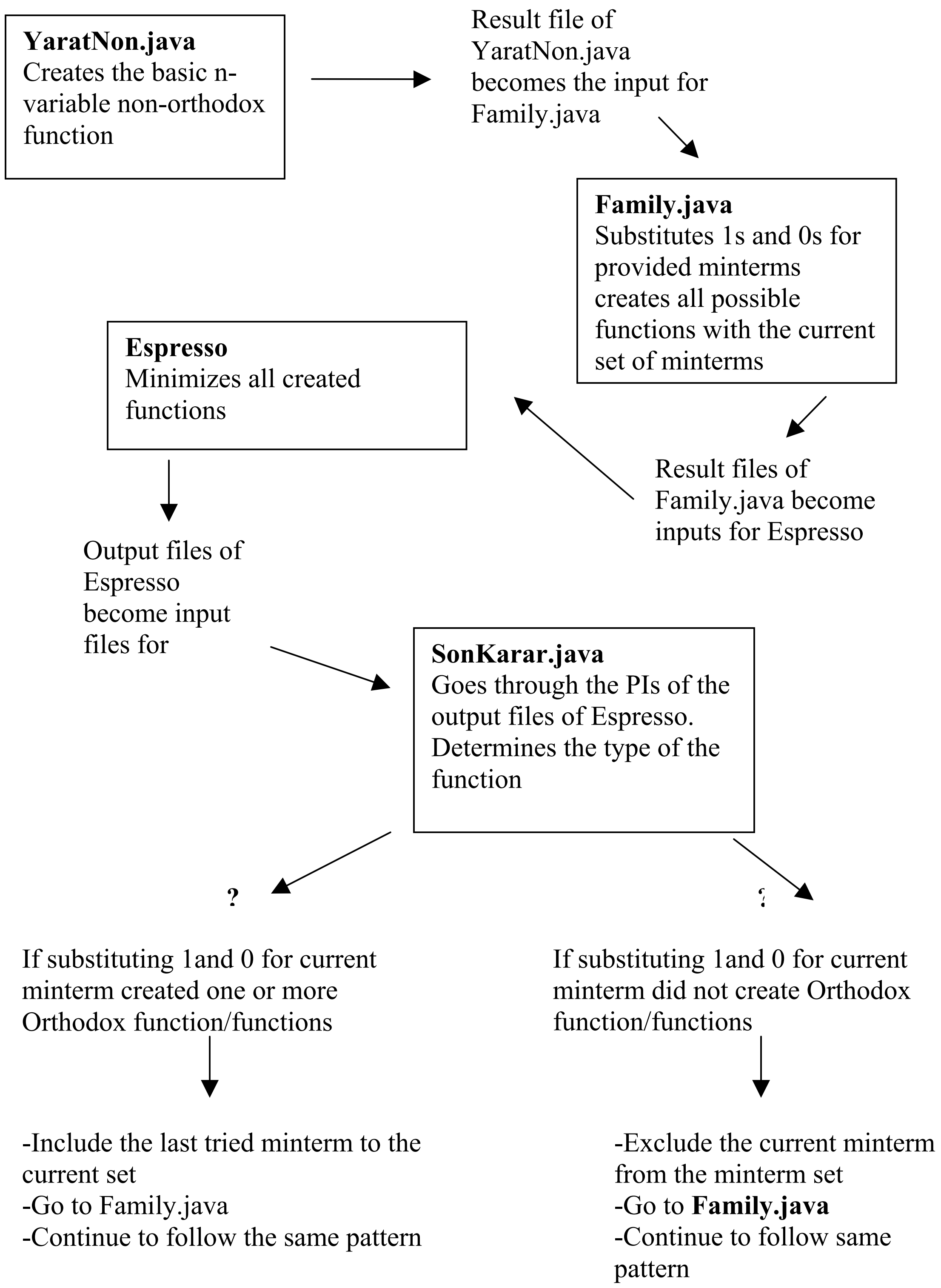
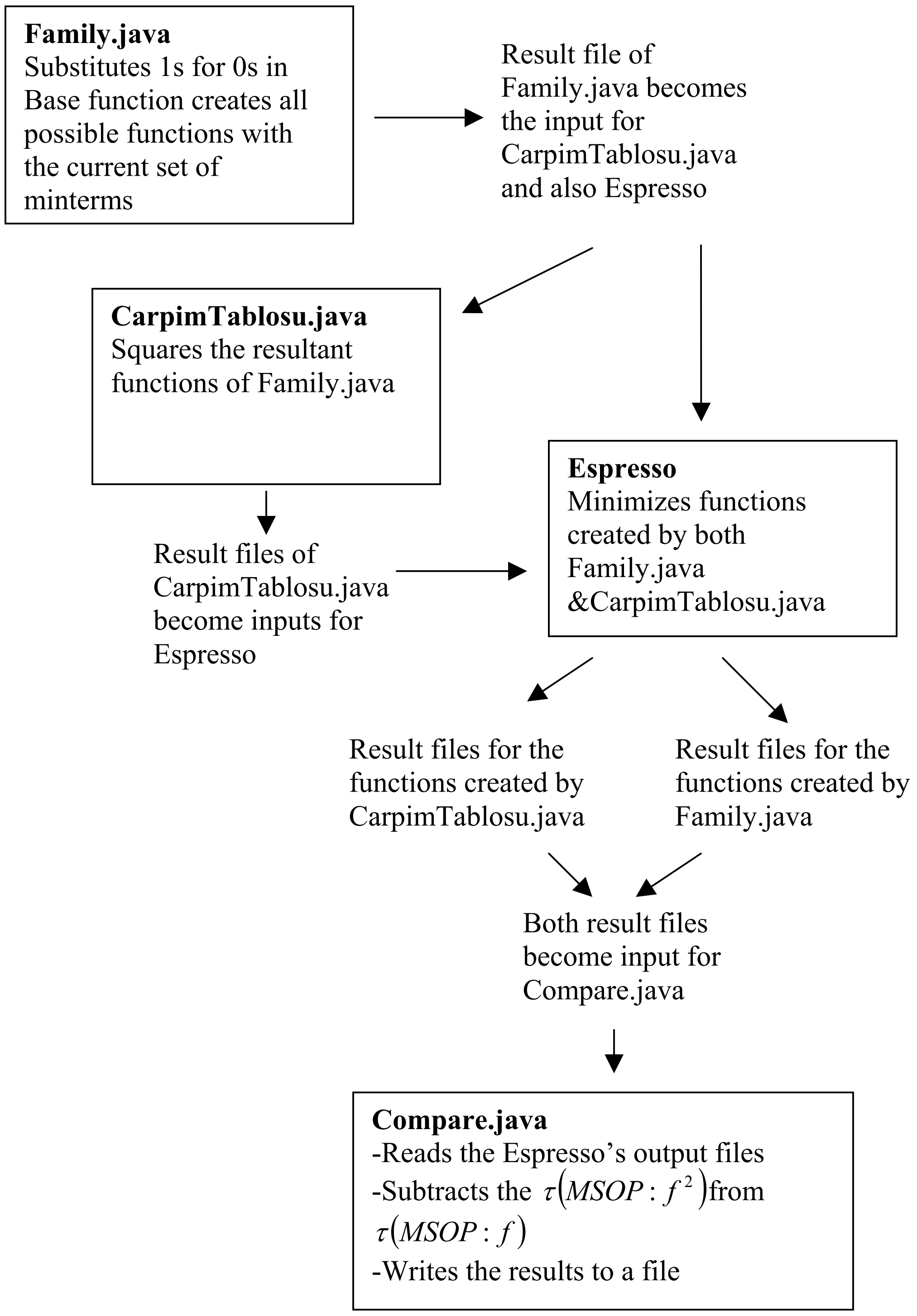

YaratNon.java
Creates the basic n-variable non-orthodox function
Result file of YaratNon.java becomes the input for Family.java
Family.java
Substitutes 1s and 0s for provided minterms creates all possible functions with the current set of minterms
Espresso
Minimizes all created functions
Result files of Family.java become inputs for Espresso
Output files of Espresso become input files for
SonKarar.java
Goes through the PIs of the output files of Espresso. Determines the type of the function
?
?
If substituting 1and 0 for current minterm created one or more Orthodox function/functions
If substituting 1and 0 for current minterm did not create Orthodox function/functions
-Include the last tried minterm to the current set
-Go to Family.java
-Continue to follow the same pattern
-Exclude the current minterm from the minterm set
-Go to Family.java
-Continue to follow same pattern

Figure 26. Usage of the java programs to determine the 6-variable non-orthodox functions.



Family.java
Substitutes 1s for 0s in Base function creates all possible functions with the current set of minterms
Result file of Family.java becomes the input for CarpimTablosu.java and also Espresso
CarpimTablosu.java
Squares the resultant functions of Family.java
Result files of CarpimTablosu.java become inputs for Espresso
Espresso
Minimizes functions created by both Family.java &CarpimTablosu.java
Result files for the functions created by CarpimTablosu.java
Result files for the functions created by Family.java
Both result files become input for Compare.java
Compare.java
-Reads the Espresso's output files
-Subtracts the $\tau(MSOP : f^2)$from $\tau(MSOP : f)$
-Writes the results to a file

Figure 27. Usage of the java programs to determine the penalty between minimization with law of distributivity and conventional minimization.

THIS PAGE INTENTIONALLY LEFT BLANK

H. OBSERVATIONS OBTAINED FROM EXPERIMENTS

The experimental analysis of the non-orthodox and orthodox functions by using the above programs and Espresso produced the following results;

1. Lemma 1

Let X be a set of variables and let y be a variable such that $y \notin X$ *. Then* $yf(X)$ *is a non-orthodox (orthodox) function if and only if* $f(X)$ *is non-orthodox (orthodox).*

For example, consider the function *f*, where $f(x_1, x_2, x_3, x_4) = \overline{x_1} x_2 \overline{x_3} + x_1 \overline{x_2}\,\overline{x_3} + \overline{x_1}\,\overline{x_2} x_3 + x_3 x_4 + x_2 x_4$. The function *f* is non-orthodox function since $\tau(MSOP : f) = 5$ and $\eta(f) = 4$ as shown in Figure 29. Logically ANDing function *f* with literal *y*, where $X = \{x_1, x_2, x_3, x_4\}$ and $y \notin X$, yields resultant function $f(X)y = \overline{x_1} x_2 \overline{x_3} y + x_1 \overline{x_2}\,\overline{x_3} y + \overline{x_1}\,\overline{x_2} x_3 y + x_3 x_4 y + x_2 x_4 y$. Figure 28 shows the Karnaugh map representation of the resultant function. It can be observed from the Figure 28 $\tau(MSOP : f(X)y) = 5$ and $\eta(f(X)y) = 4$. The minterms marked with "✧" in Figure 28 are the ones that belong to the maximum independent minterms set, and the circled minterms represent the PIs of the resultant function.

To prove Lemma 1, following idea can be used. A PI, denoted as *p*, of function $f(X)$ has the property that py is a PI of $f(X)y$. Similarly, a PI of $f(X)y$ has the form *py*, where *p* is a PI of $f(X)$. As a result, it is obvious that logically ANDing a function with a literal that does not belong to its variable set, is nothing more than increasing the number of variables of the original function by 1, and all the properties of the function, like $\tau(MSOP : f)$ and $\eta(f)$ remain unchanged.

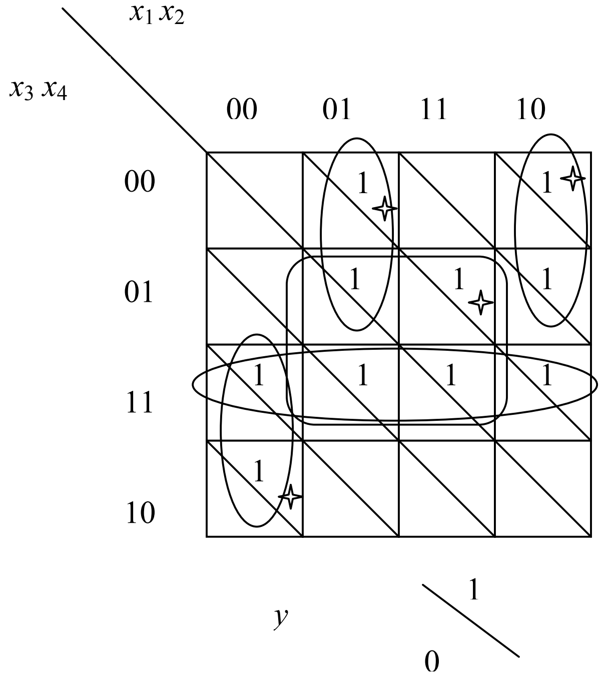


Figure 28. Karnaugh map representation of $f(X)y$.

2. Lemma 2

Let X be a set of variables and let y be a variable such that $y \notin X$. *Then* $y \vee f(X)$ *is a non-orthodox function if and only if* $f(X)$ *is non-orthodox.*

To demonstrate this lemma, Lemma 1's example function has been used. Once again, function f has the non-orthodox property since $\tau(MSOP : f) = 5$ and $\eta(f) = 4$. Figure 29 depicts the original function $f(X)$'s Karnaugh map. The minterms marked by ✧ belong to the maximum independent minterms set, and the circled minterms show the PIs that belong to MSOP of function *f*.

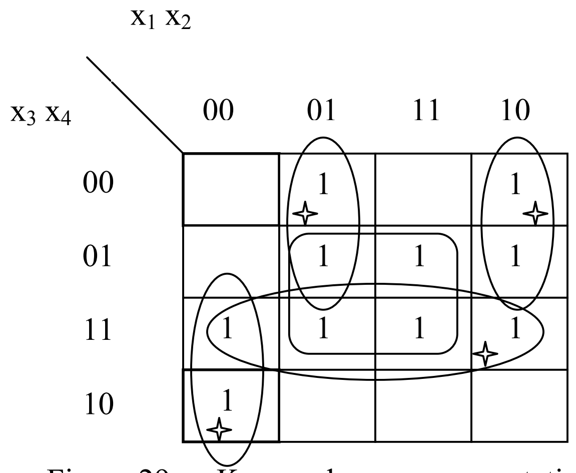


Figure 29. Karnaugh map representation of
$f(x_1, x_2, x_3, x_4) = \overline{x_1} x_2 \overline{x_3} + x_1 \overline{x_2 x_3} + \overline{x_1 x_2} x_3 + x_3 x_4 + x_2 x_4$.

Logically ORing the function *f* with the literal *y*, where $y \notin X$, yields following function, that is; $f(X) \vee y = x_1 x_2 \overline{x_3} + x_1 \overline{x_2 x_3} + \overline{x_1 x_2} x_3 + x_3 x_4 + x_2 x_4 + y$, which can be represented as we shown in Figure 30. Again, the minterms that are marked by "✧ " belong to the maximum independent minterms set, and the circled minterms show the MSOP of function *f*.

As it can be seen from Figure 30, $\tau(MSOP : f(X) \vee y) = 6$ and $\eta(f(X) \vee y) = 5$. Thus, the resultant function is also a non-orthodox function.

To prove Lemma 2, the idea used in Lemma 1 has been used. A PI, denoted as *p*, of function *f* has the property that p is a PI of $f \vee y$. Similarly a PI of $f(X) \vee y$ has the form $p \vee y$, where p is a PI of $f(X)$. So, it can be observed that to logically OR a function with a literal that does not belong to its variable set means to increase the number of variables of the original function and add 1 more essential PI to the MSOP representation of it.

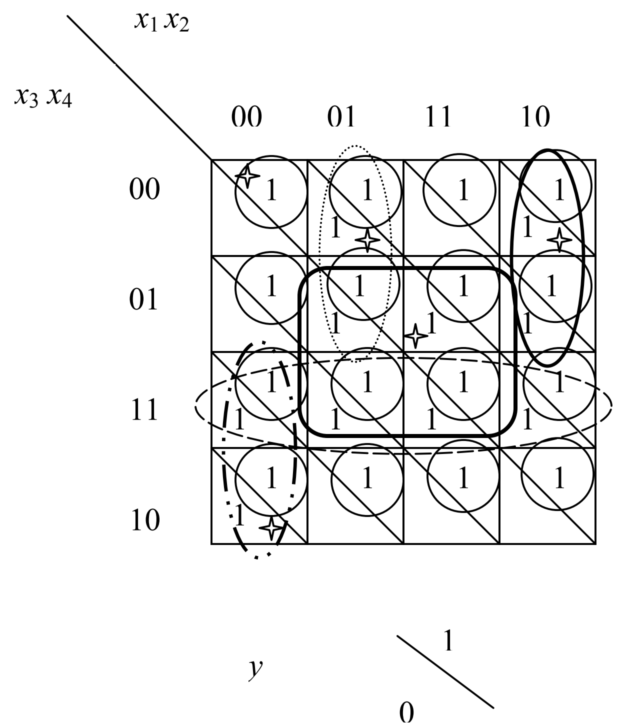


Figure 30. Karnaugh map representation of $f(X) \vee y$.

The converse also holds. That is, if the given function has the orthodox property, the resultant function will also hold the orthodox function property when it is logically ORed with a literal, where the literal does not belong to the variable set of the function.

3. Lemma 3

Let X be a set of variables and let y be a variable such that $y \notin X$. Then $f(X) \oplus y$ is a non-orthodox function if $f(X)$ is non-orthodox.

The proof of Lemma 3 proceeds as follows;

Following can be written; $f(X) \oplus y = \overline{f}(X)y + f(X)\overline{y}$, and let p be a PI of $f(X) \oplus y$. It follows that either one of the followings is true;

1. $pf(X)\overline{y} = p$ and $p\overline{f}(X)y = 0$ or
2. $pf(X)\overline{y} = 0$ and $p\overline{f}(X)y = p$.

Thus, the PIs of $\overline{f}(X)y$ and $f(X)\overline{y}$ are disjoint. Similarly, the minterms in the maximal independent set of $f(X) \oplus y$ form two disjoint subsets, one with $y = 0$ and the other with $y = 1$. The former represents a maximal independent set of $f(X)\overline{y}$ and the latter of $\overline{f}(X)y$.

Since $f(X)$ is non-orthodox, the number of the PIs in its MSOP is greater than the number of minterms in its MIS. The same is true for $f(X)\overline{y}$. Thus, it follows that $f(X) \oplus y$ is non-orthodox.

Consider following example. First, it is shown that PIs of $\overline{f}(X)y$ and $f(X)\overline{y}$ are disjoint. Let $f(x_1, x_2) = \overline{x_1} + x_2$. Logically EXORing this function with literal y, where $y \notin X$, yields a function that is $z(x_1, x_2, y) = \overline{x_1}y + x_2 y + x_1\overline{x_2}y$. Figure 31 shows the Karnaugh map representations of $f(X)$ and $z(x_1, x_2, y)$,as it is seen from the figure the PIs of subset $\overline{f}(X)y$ and $f(X)\overline{y}$ are disjoint.

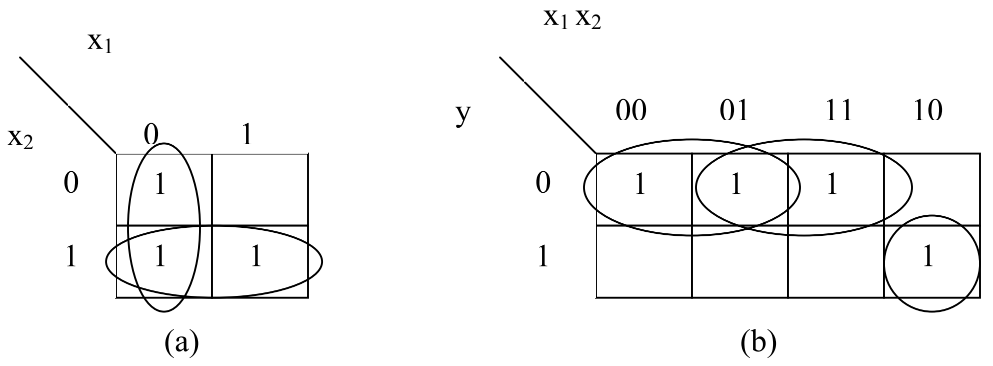


Figure 31. (a) Shows the PIs of the function f (X) (b) Shows the PIs of function f (X) ⊕ y.

4. Lemma 4

The following table is obtained from the experiments.

$f(X)$	$g(Y)$	$f(X) \vee g(Y)$ where $X \cap Y = 0$
Orthodox	Orthodox	Orthodox
Orthodox	Non-orthodox	Non-orthodox
Non-orthodox	Orthodox	Non-orthodox
Non-orthodox	Non-orthodox	Non-orthodox

Table 10. Type of the resultant function obtained from the logical OR of two disjoint functions.

It follows;

$f(X) \vee g(Y)$, where $X \cap Y = 0$, is non-orthodox if and only if $f(X)$ or $g(Y)$ or both is non-orthodox.

The proof of Lemma 4 proceeds as follows;

Because $X \cap Y = 0$, there is a one-to-one correspondence can be established between minterms in the maximal independent sets of $f(X)$ and $g(Y)$ and the maximal independent set of $f(X) \vee g(Y)$. Similarly, a one-to-one correspondence can be established between PIs of $f(X)$ and $g(Y)$ and PIs of $f(X) \vee g(Y)$. If the size of the maximal independent set of $f(X)$ or $g(Y)$ exceeds the size of the minimum sum-of-products of $f(X)$ or $g(Y)$, respectively, the same will be true of $f(X) \vee g(Y)$. This proves the (if) part. The proof of the (only if) part is proved in a similar way.

5. Lemma 5

Let f be a self dual function. Then f is non-orthodox if and only if $\overline{f}$ is non-orthodox.

The proof of the Lemma 5 proceeds as follows.

For all functions *f*, $f(x_1,x_2,...,x_n)$ is non-orthodox if and only if $f(\overline{x_1},\overline{x_2},...,\overline{x_n})$ is non-orthodox. That is complementing variables does not affect the orthodox property of a function (the PIs and minterms in the maximal independent set differ only by a complementation of the variables). In a self-dual function, $f(\overline{x_1},\overline{x_2},...,\overline{x_n})=\overline{f}(x_1,x_2,...,x_n)$. Thus, it follows that $f(x_1,x_2,...,x_n)$ is non-orthodox if and only if $\overline{f}(x_1,x_2,...,x_n)$ is non-orthodox.

The significance of Lemma 5 is that it identifies a set of functions with a special property. That is, for self-dual function *f*, either

1. both f and $\overline{f}$ are orthodox or
2. both f and $\overline{f}$ are non-orthodox.

The self dual function $S_{2,3}(x_1,x_2,x_3)$ in Figure 13 is symmetric. Thus, it is orthodox. Its complement, which is also symmetric, is also orthodox. There remains the question of whether there exists self-dual functions that are non-orthodox. Indeed, there are.

Consider the 5-variable function.

$$g(x_1,x_2,x_3,x_4,x_5)=\overline{x_5}h(x_1,x_2,x_3,x_4)\vee x_5\overline{h}(x_1,x_2,x_3,x_4)$$

where $h(x_1,x_2,x_3,x_4)$ is the 4-variable counterexample that is used by Sasao and Butler [1] shown in Figure 15 in Chapter III. g is non-orthodox, as can be seen from Figure 33 (a). It is shown that g is self dual by showing $g(\overline{x_1},\overline{x_2},...,\overline{x_n})=\overline{g}(x_1,x_2,...,x_n)$. First,

$$g(\overline{x_1},\overline{x_2},\overline{x_3},\overline{x_4},\overline{x_5})=x_5h(\overline{x_1},\overline{x_2},\overline{x_3},\overline{x_4})\vee\overline{x_5}\,\overline{h}(x_1,x_2,x_3,x_4)\,(1)$$

second,

$$\overline{g}(x_1,x_2,x_3,x_4,x_5)=\{x_5\vee\overline{h}(x_1,x_2,x_3,x_4)\}\{\overline{x_5}\vee h(\overline{x_1},\overline{x_2},\overline{x_3},\overline{x_4})\}$$

$$=\overline{x_5}\,\overline{x_5}\vee\overline{x_5}\,\overline{h}(x_1,x_2,x_3,x_4)\vee x_5h(\overline{x_1},\overline{x_2},\overline{x_3},\overline{x_4})$$

$$\vee\,\overline{h}(x_1,x_2,x_3,x_4)h(\overline{x_1},\overline{x_2},\overline{x_3},\overline{x_4})\quad(2)$$

$$= \overline{x_5}\, h(x_1, x_2, x_3, x_4) \vee x_5 h(\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}) \quad (3).$$

The last term in (2) is a consensus term. Any assignment of values to x_1, x_2, x_3 and x_4, that make it 1 also cause exactly one of the two middle terms in (2) to be 1. Since (2) and (3) are identical, we have identified a non-orthodox self dual function.

The significance of this result is the establishment of the existence of a non-orthodox function, whose complement is also non-orthodox.

6. Observation 1

As it is explained earlier, the counterexamples, that have been used by Voight and Wegner [5] in 1989 and Sasao and Butler [1] in February 2001, to prove that Disjoint Computation Scheme Hypothesis for logical AND does not always hold, are not totally different functions. In reality they are closely related to each other. The reason that leads us to this observation is Lemma 1.

The 4-variable counterexample that used by Sasao and Butler [1], is $f(x_1, x_2, x_3, x_4) = \overline{x_1} x_2 \overline{x_3} + x_1 \overline{x_2}\,\overline{x_3} + \overline{x_1}\,\overline{x_2} x_3 + x_3 x_4 + x_2 x_4$. It is part of one family of non-orthodox functions that has 2 don't cares. As explained in Chapter 3, by substituting 0s and 1s, representatives of the 4 NP-equivalent classes can be determined. To transform Voight and Wegner's example to that of Sasao and Butler, substitute a 1 for don't care in minterm 1110 and a 0 for don't care in minterm 0000 in Sasao and Butler's counterexample. This yields $f(x_1, x_2, x_3, x_4) = x_1 \overline{x_2}\,\overline{x_3} + \overline{x_1}\,\overline{x_2} x_3 + x_1 x_2 x_3 + x_3 x_4 + x_2 x_4$. If we logically AND this function with x_5, where $x_5 \notin X$, we obtain a 5-variable non-orthodox function, $x_1 \overline{x_2}\,\overline{x_3} x_5 + \overline{x_1}\,\overline{x_2} x_3 x_5 + x_1 x_2 x_3 x_5 + x_3 x_4 x_5 + x_2 x_4 x_5$.

Interchanging $\overline{x_4}$ with x_5, yields Voight and Wegner's counterexample; $f(x_1, x_2, x_3, x_4, x_5) = \overline{x_1} x_2 \overline{x_3}\,\overline{x_4} + x_1 \overline{x_2}\,\overline{x_3}\,\overline{x_4} + \overline{x_1}\,\overline{x_2} x_3 \overline{x_4} + x_1 x_2 x_3 \overline{x_4} + x_3 \overline{x_4}\,\overline{x_5} + x_2 \overline{x_4}\,\overline{x_5}$.

This is shown in Figure 32. As a result, Voight and Wegner's counterexample can be obtained by logically ANDing Sasao and Butler's [1] NP_{01} with variable x_5.

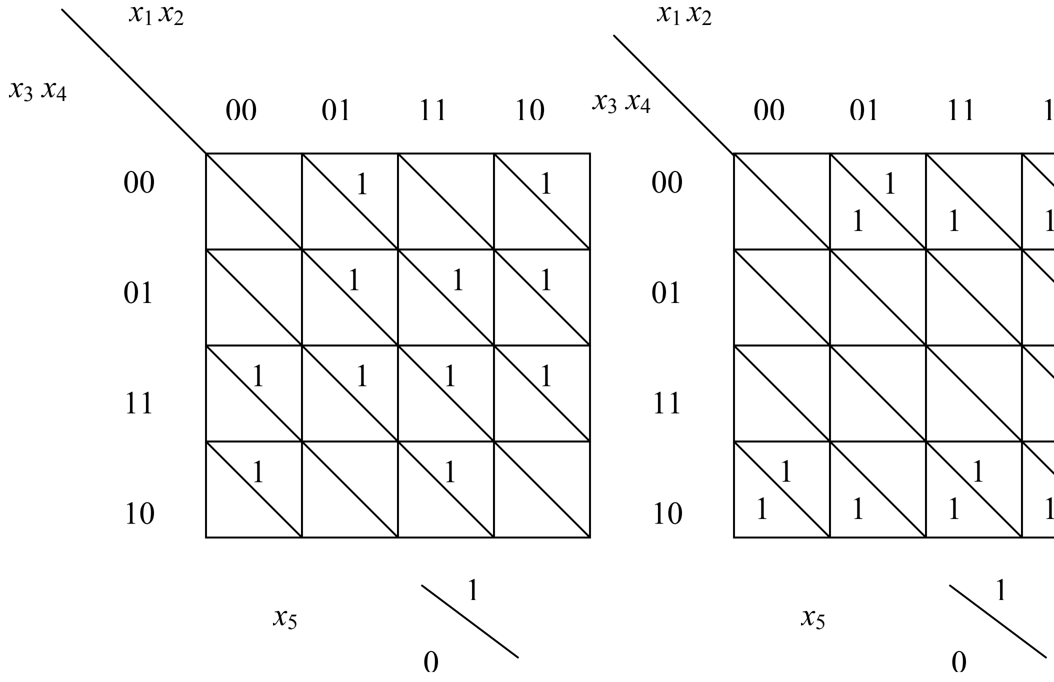


Figure 32. (a) Minterms of x_5 NP_{01} (b) Minterms of Voight and Wegner's [4] counterexample.

7. Observation 2

After conducting experiments with 256 4-variable and chosen 20 5-variable, and 20 6-variable non-orthodox functions, it has been observed that complementation of the non-orthodox functions yields almost always orthodox functions, results of the complementations are shown in Table 11. All the ones that yield non-orthodox function were self-dual functions and they were sharing a common point. So that, one can make an observation that *for 4, 5 and 6-variable non-orthodox functions complementing the given non-orthodox function yields an orthodox function except a specific subset of self-dual functions.*

The following explains how a special self dual function can be constructed with non-orthodox function property that yields an orthodox function when it is complemented.

Types and numbers of the functions	Resultant function is orthodox	Resultant function is non-orthodox
256 4-variable non-orthodox functions	256	None
20 5-variable functions	19	1
20 6-variable functions	17	3

Table 11. Results for complementation of the chosen non-orthodox functions.

As in SonKarar.java's proof, a base function *f* that is created by YaratNon.java can written as follows;

(1) $f = \left(S_{1,2,\ldots,\frac{n}{2}-1} \right) \overline{x_n} \vee (x_1 \vee x_2 \vee \ldots \vee x_{n-1}) x_n$, *n* is the number of the variables of the base non-orthodox function, where *n* is even.

It is possible to create a non-orthodox function such that complementing it yields a non-orthodox function also, by doing the following steps;

- Determine the minterms of *f* by using (1).
- Form a set from all possible minterms with n-variables. Then, remove the ones that belong to *f* from this set.
- Promote the remaining ones to (*n+1*)-variable minterms by appending a 1 to their cube notations.
- Assign a 1 to each of these newly created minterms.
- Promote the *f*'s minterms to (*n+1*)-variable minterms by appending a 0 to their cube notations.
- Assign a 1 to each of these newly created minterms.

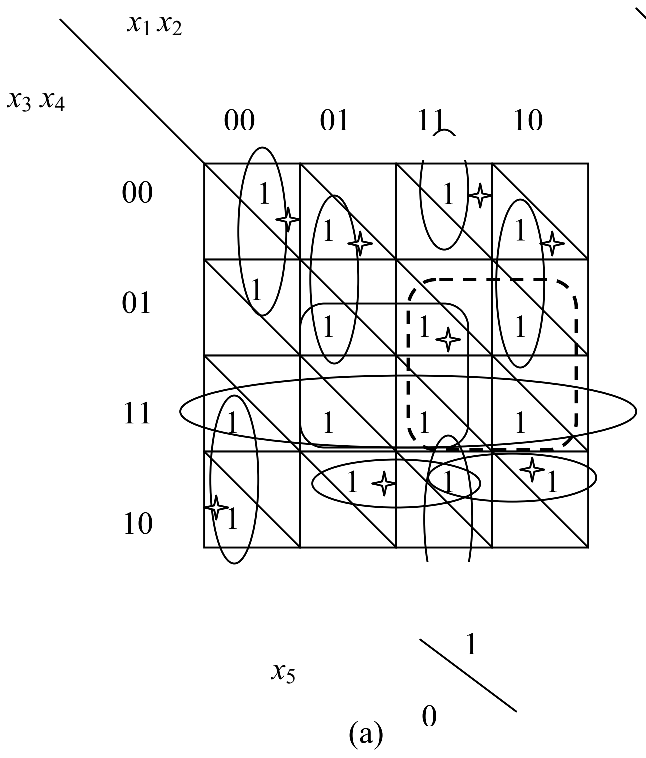
Consider *f*, where $f(x_1,x_2,x_3,x_4) = x_1x_2\overline{x_3} + x_1\overline{x_2}\,\overline{x_3} + \overline{x_1}\,\overline{x_2}x_3 + x_3x_4 + x_2x_4$ as an example. *f* is a non-orthodox function since $\tau(MSOP : f) = 5$ and $\eta(f) = 4$. If we apply the above steps to *f* following will be the resultant function;

$$f_1(x_1,x_2,x_3,x_4,x_5) = \overline{x_1}x_2\overline{x_3}\,\overline{x_5} + x_1\overline{x_2}\,\overline{x_3}\,\overline{x_5} + \overline{x_1}\,\overline{x_2}x_3\overline{x_5} + x_3x_4\overline{x_5} + x_2x_4\overline{x_5}$$
$$+ \overline{x_1}\,\overline{x_2}\,\overline{x_3}x_5 + x_2x_3\overline{x_4}x_5 + x_1x_3\overline{x_4}x_5 + x_1x_2\overline{x_4}x_5 .$$

*f*1 is also a non-orthodox function since, $\tau(MSOP : f_1) = 9$ and $\eta(f) = 8$ as shown in Figure 33(a), if we complement *f*1 resultant function, $\overline{f_1}$ becomes following;

$$\overline{f_1}(x_1,x_2,x_3,x_4,x_5) = (x_1 + x_2 + x_3 + x_5)(x_1 + \overline{x_2} + x_3 + \overline{x_5})(\overline{x_1} + \overline{x_2} + x_4 + x_5)$$
$$(\overline{x_1} + x_2 + x_3 + \overline{x_5})(\overline{x_2} + \overline{x_4} + \overline{x_5})(\overline{x_3} + \overline{x_4} + \overline{x_5})(x_1 + x_2 + \overline{x_3} + \overline{x_5})(\overline{x_2} + \overline{x_3} + x_4 + x_5)$$
$$(\overline{x_1} + \overline{x_3} + x_4 + x_5).$$

$\overline{f_1}$ is a non-orthodox function since, $\tau(MSOP : \overline{f_1}) = 9$ and $\eta(\overline{f_1}) = 8$ as shown in Figure 33 (b).



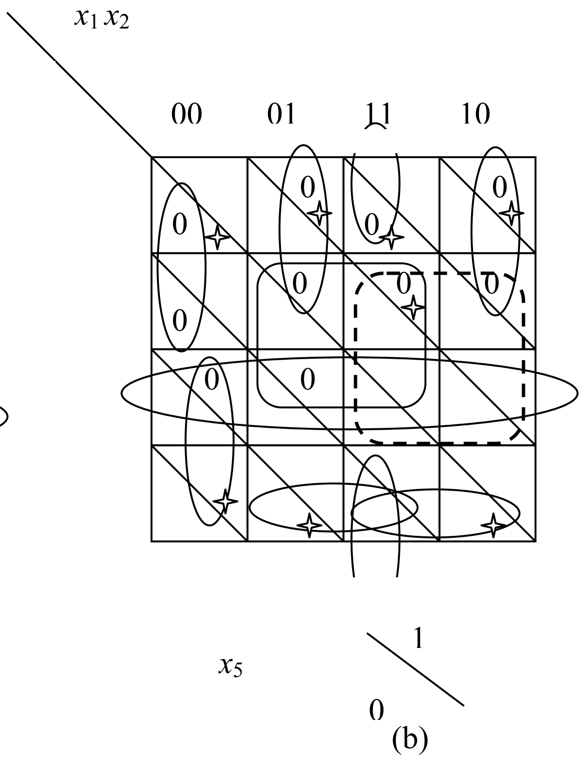


Figure 33. (a) Karnaugh map representation of *f*1 (b) Karnaugh map representation of $\overline{f_1}$.

8. Conjecture 1

It is known that the number of the non-orthodox functions grows dramatically as for the number of variables increases [1]. Since there are many non-orthodox functions, the following conjecture seems reasonable.

The probability *P*(*f*) that an arbitrary n-variable function *f* and its complement are non-orthodox approaches 1 as *n* approaches infinity.

Indeed, if Conjecture 1 is false, it is likely that approximately one-half of n-variable functions, for large n, are orthodox and one-half are non-orthodox. This does not seem to be case, as experimental results by Sasao and Butler show [1].

9. Conjecture 2

Functions can be split into two major groups; non-orthodox and orthodox functions. One may think about the properties of the resultant functions that obtained by logical operations of these non-orthodox and orthodox functions.

As n (the number of variables) increases, the number of the functions increases significantly, 2^{2^n}. So, that when we increase the number of the variables, it becomes hard to analyze all the functions, since there are so many functions. It is better to randomly pick functions among the non-orthodox and orthodox functions and perform the logical operations by using these functions and by the help of these operations results develop some conjectures.

What follows are a number of conjectures that have been developed by the help of experimental analysis.

Let $f(X)$ be a non-orthodox function on variable set X, and let $g(Y)$ be any function on variable set Y, such that $X \cap Y = 0$. Then $f(X) \wedge g(Y)$ is a non-orthodox function.

After conducting logical AND operations with 256 4-variable non-orthodox functions, randomly picked 25 3-variable, 25 4-variable, and 25 5-variable orthodox functions, the above statement observed. A java program, Espresso2.java, conducted the logical ANDing operations. Table 12 shows result of the AND operations between non-orthodox and orthodox functions.

Function	Operation	Function	Result
256 4-variable non-orthodox	AND	25 3-variable orthodox	All results are non-orthodox
256 4-variable non-orthodox	AND	25 4-variable orthodox	All results are non-orthodox
256 4-variable non-orthodox	AND	25 5-variable orthodox	All results are non-orthodox

Table 12. Results for logical AND operation between all 4-variable non-orthodox functions and randomly chosen orthodox functions.

10. Non-orthodox Functions with 2n-variable

Chapter 3 explains how to obtain non-orthodox functions with 2n-variables, where $n = 1,2,3,\ldots,\infty$.

a. 6-variable Non-orthodox Functions

Algorithm 3 creates only one non-orthodox function for each chosen *n*, where these functions are base functions. They are called base functions since they are the simplest known functions for their variable sets. And, each of them can be written as follows.

$$f = \left(S_{1,2,\ldots,\frac{n}{2}-1} \right) \overline{x_n} \vee \left(x_1 \vee x_2 \vee \ldots \vee x_{n-1} \right) x_n .$$

Let *n*=3, then the function that is dealt with a 6-variable function. Figure 25 shows a non-orthodox function that is created by Algorithm 3 for 6-variable functions. After the creation of the base non-orthodox function, exploration of remaining 6-variable non-orthodox function started. To do this, two Java programs used, namely Family.java and SonKarar.java as explained in “The usage of programs” part. Lack of knowledge in 6-variable non-orthodox functions, forced the research to a brute force approach during the investigation process of 6-variable functions. It is known that there are 64 cells (minterms) in a 6-variable function’s Karnaugh map and Algorithm 3 showed that 46 of them are 1 as seen in Figure 25. So, Family.java used the remaining 18 minterms as don’t cares and created every possible 6-variable function by adding these don’t cares to the minterms set of the base 6-variable non-orthodox function. Then, SonKarar.java made a decision for each created function’s type.

At the end of the process, two different representatives for the 6-variable non-orthodox functions have been found, where each of them is an incompletely specified function. Figure 34 shows the first representative function. It has 13 don’t cares, which suggests $2^{13} = 8192$ completely specified functions.

To be able to obtain a function, the reader needs to assign values to the dashes in Figure 34, where each dash represents a don’t care.

Figure 35 shows the second representative function for the 6-variable non-orthodox functions, this representative function has 14 dashes (don’t cares), which suggests $2^{14} = 16384$ completely specified functions.

The representative function seen in Figure 35 has the minterms 110110, 101110 and 011010 as don’t cares, where these same minterms are 0s in the representative function seen in Figure 34.

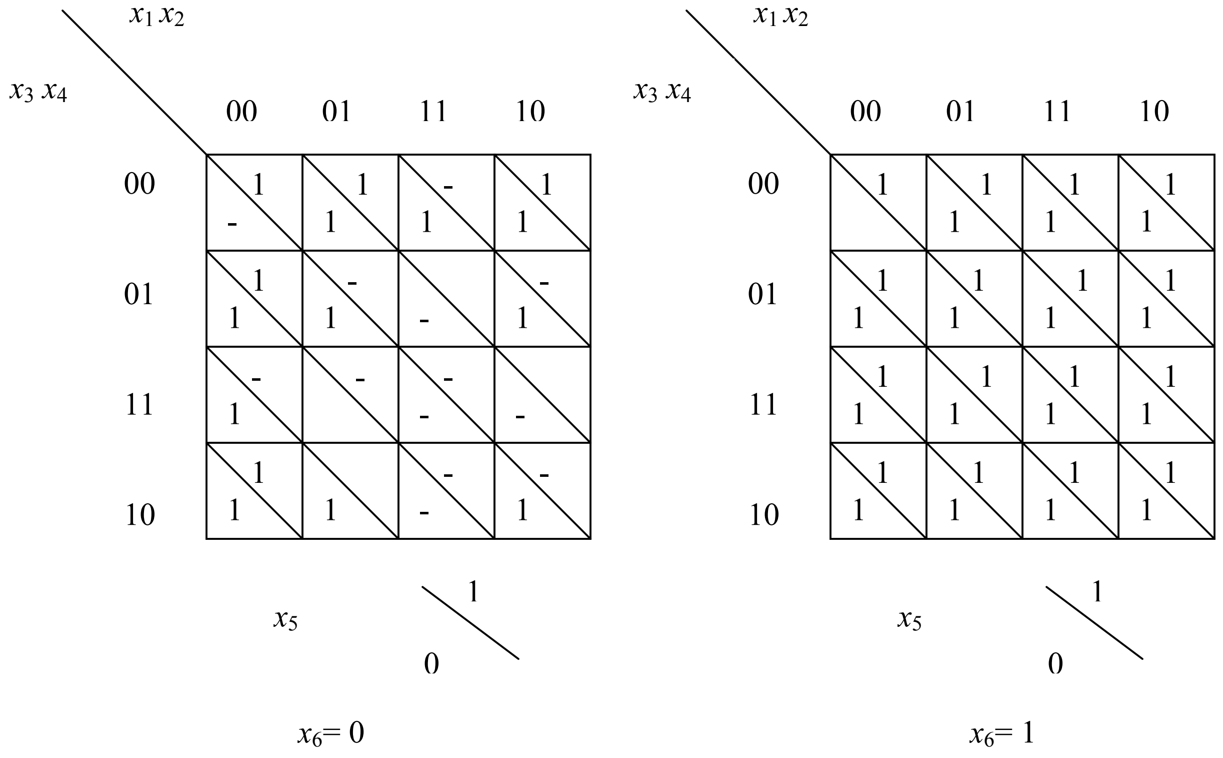


Figure 34. A 6-variable non-orthodox function with 13 don’t cares.

During the first attempt to determine the representative of the 6-variable non-orthodox functions, minterm 001110 was picked as a don’t care. SonKarar.java verified that it could be used as a don’t care. Later the process depicted in Figure 26 repeated by using the other potential don’t care minterms. But, in the second attempt, minterm 001110 was kept as a 0, not a don’t care, and determining the other 6-variable non-orthodox functions continued with remaining potential don’t care minterms. At the end of the first attempt, 13 don’t cares have been determined. Thus, the number of the non-orthodox functions is 8192. And, at the end of the second attempt 14 don’t cares have been determined. Thus, the number of the non-orthodox functions is 16384.

Now the question “Are there more representatives with different numbers of don’t cares?” might arise.

It is believed that the 16384 6-variable completely sepecified functions corresponding to the incompletely specified function in Figure 35 are all non-orthodox functions. That is Espresso produces minimal solutions that have more PIs then what is believed to be the number of minterms in the maximal independent set. It is true for Figure 34 also.

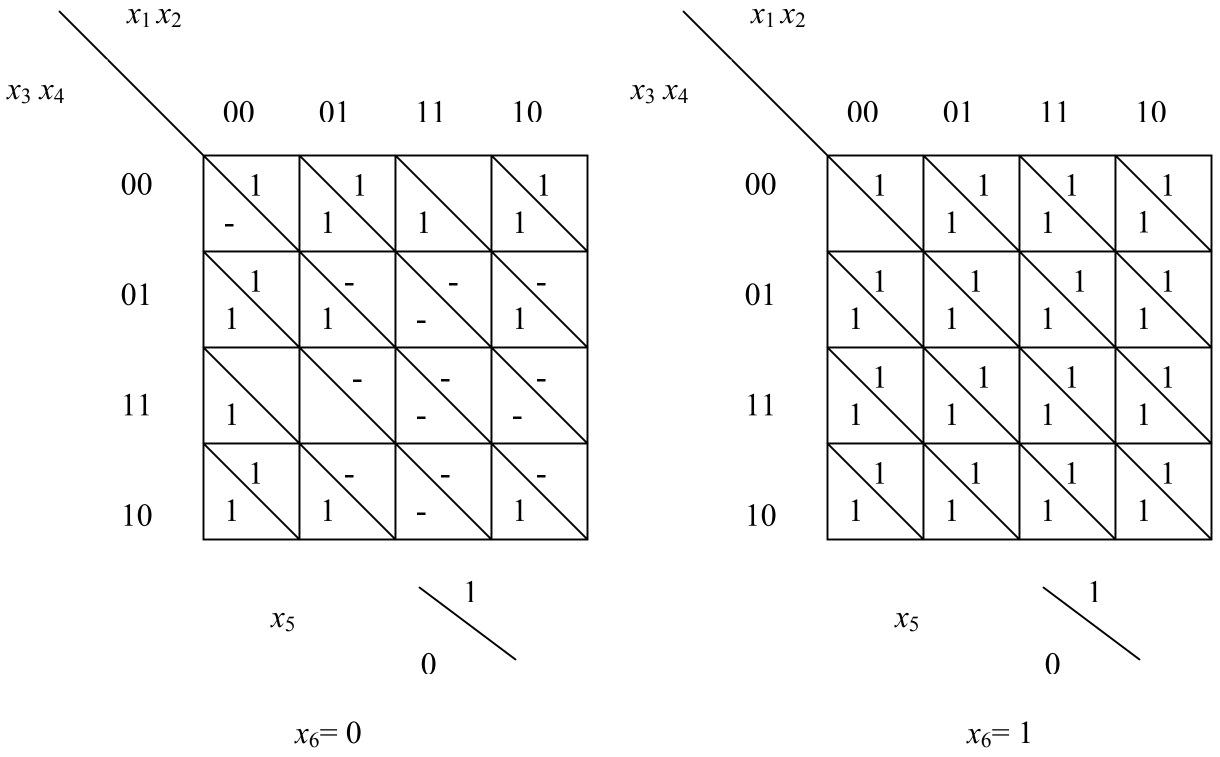


Figure 35. A 6-variable non-orthodox function with 14 don’t cares.

b. Simplification by Applying the Law of Distrubitivity and Without Applying the Distrubitivity

The computer program that is used by Sasao and Butler [1] verified that the number of the PIs in the MSOPs of the 4-variable non-orthodox functions is always 1 more than the number of the minterms in their maximal independent sets.

This same program also verified that squaring a 4-variable non-orthodox function, *f*, by logically ANDing two copies of *f* and using disjoint variable sets for each copy, to obtain an 8-variable AND bi-decomposable function, f^2, yields 1 more PI than the number of the PIs yielded by minimization of f^2 when the law of distributivity is applied for the minimization.

It has been observed from results of the experiments described in this thesis, on the contrary to 4-variable non-orthodox functions, the number of the PIs in the

MSOPs of 6-variable non-orthodox functions is greater than the number of the minterms in their maximal independent minterms sets by 2 or 1.

Consider the question, “What is the penalty of minimizing the 12-variable AND decomposable functions by applying the distributivity law?” (12-variable functions obtained from 6-variable non-orthodox function). Unfortunately, during the minimization of the 12-variables functions it was not possible to use the flag –Dexact with Espresso because of excessive computation time. Table 13 shows the computation times of Espresso with or without –Dexact flag for various numbers of variables.

During the experiments with the 12-variable functions a total of 16384 functions are analyzed. Since, the –Dexact flag could not used for the simplification of the 12-variable functions, the question asked above cannot be answered precisely. Despite this fact, results of the experiments may give a rough idea about the range of the difference between $\tau(MSOP : f^2)$, minimizing the functions without breaking them into subfunctions, and $\tau(MSOP : f)^2$ minimizing the functions by using the law of distributivity.

When 6-variable non-orthodox functions are squared to obtain AND bi-decomposable 12-variable functions, unlike 4-variable non-orthodox functions, the difference between the number of PIs in MSOPs that obtained by applying law of distributivity and the number of PIs in MSOPs that obtained from application of a conventional minimizing approach is not fixed to 1. But, the difference between $\tau(MSOP : f^2)$ (conventional approach for minimization) and $\tau(MSOP : f)^2$ (divide-and-conquer approach for minimization) varies from 0 to 19. The worst case, 19 PIs difference, has been encountered 43 times during the experiments, which yields a percentage of 0.026%. Avarage number of the PIs in MSOPs of these 43 12-variable AND bi-decomposable functions were 207 that yields a percentage of 9% for the difference (19 PIs difference), which was 4% (1 PI) for the worst case of 8-variable AND bi-decomposable functions [1].

Number of variables	Minimization with –Dexact flag (in milliseconds)	Minimization without –Dexact flag (in milliseconds)
4	62	62
5	68	67
6	74	72
7	73	72
8	2948	80
9	2072	82
10	3397	76
11	3998	178
12	No result. (after a week run time)	732

Table 13. Average requirement computation time for minimization with Espresso.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

Determining the properties of non-orthodox and orthodox functions and determining which type of functions are orthodox and which type of functions are non-orthodox are not completed yet. In this immense research area, this thesis opens only a small window and tries to show the importance of the orthodox functions in logical design.

From the results of this thesis, the following can be stated:

1. Logically ANDing a non-orthodox (orthodox) function with a literal yields a non-orthodox (orthodox) function.
2. Logically ORing a non-orthodox function with a literal yields a non-orthodox function.
3. Logically EXORing a non-orthodox function with a literal yields a non-orthodox function.
4. Complementing a non-orthodox function of 4 or 6-variables yields an orthodox function except the self-dual non-orthodox functions (from experimental evidence).
5. Logically ORing two functions on disjoint sets of variables yields a non-orthodox function if and only if one of the two functions or both of them non-orthodox, and yields an orthodox function if and only if both of them orthodox functions.
6. Logically EXORing two functions on disjoint set of variables yields a non-orthodox function if and only if one or both of the two functions are non-orthodox, and yields an orthodox function if and only if both of them are orthodox functions.
7. Logically ANDing two functions on disjoint sets of variables yields a non-orthodox function if and only if one or both of the two functions are non-orthodox, from experimental evidence.

8. It is shown that the counterexample that was proposed by Voight and Wegner [6], is closely related with Sasao and Butler's [1] counterexample. So, it is the simplest known non-orthodox function.
9. The penalty paid when one uses the law of distributivity to minimize the functions with AND bi-decomposition property, where each subfunction is non-orthodox, grows when the number of the variables of the function grows. (From experimental results, it can go up to 19 PIs for a 12-variable AND bi-decomposable function, but this result is not certain as explained in the "6-variable non-orthodox functions" part of Chapter V).
10. Two representative functions are proposed. One of them has 13 don't cares and the other one has 14 don't cares. They show all 6-variable non-orthodox functions that were discovered during the experitmental research in a compact form. A 6-variable non-orthodox function can be obtained from these representatives by assigning values to the don't cares. Unfortunately, not all 6-variable non-orthodox functions were discovered.

Also the experimental results of this thesis developed an understanding of the importance of orthodox functions in the minimization process of practical functions. That is, if a practical function can divided into two component functions (bi-decomposition property) and each of the components are orthodox, then minimizing the componenets separately and applying the law of distributivity yields MSOP for the practical function (the divide-and conquer algorithm). Note that this algorithm gives us an improved computation time with respect to conventional minimization algorithms [1].

Determining the types of the components may help us to choose the minimization approach also. If either one of the component functions or both of them are non-orthodox, then choosing divide-and-conquer algorithm cause us to pay a penalty (more PIs in the MSOP expression then that needed) that tends to increase when the variable number of the component functions increase. Thus, excepting the cases that both of the component functions are orthodox, a conventional minimizing approach should be used.

B. FUTURE RESEARCH RECOMMENDATIONS

Note that this research only considers the conceptual perspective of the orthodox and non-orthodox functions. That is, it tries to address the question, “What are the properties of these functions?” rather than using the orthodox and non-orthodox functions in a logical design, creating a prototype of this design and testing it.

Subsequent research may take the presented properties and apply them in a logical design or may create a minimization tool that applies the following minimization algorithm (this algorithm is proposed by Sasao and Butler [1]).

Algorithm 4

1. If f has an OR bi-decomposition, then minimize the SOPs of each component function separately. The OR of two MSOPs gives an MSOP for f.
2. If f has an AND bi-decomposition, determine the types of the component functions, orthodox or non-orthodox. If both are orthodox minimize, them separately apply the law of distributivity to derive the MSOP for f.
3. Otherwise, use a conventional approach to minimize f.

Although the proposed algorithm in Chapter IV of this thesis, Algorithm 3, can be used to construct n-variable non-orthodox functions, where n is even, because of the large number of functions and their excessive computation time, this thesis focused on 4 and 6-variable non-orthodox functions and their families to address the characterization problem of non-orthodox functions. A follow-on research may propose another algorithm that overcomes this large-number-of-functions problem by creating randomly chosen non-orthodox functions and continue to characterize the orthodox and non-orthodox functions.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Sasao, T. and Butler, T.B., *On the Minimization of SOPs for Bi-Decomposable Functions*, Design Automation Conference, Proceedings of the ASP-DAC, pp. 219-224, 2001.

[2] Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill Computer Science Series, 1970.

[3] Sasao, T. and Butler, T.B. *On Bi-Decomposoitions of Logic Functions*, International Workshop on Logic Synthesis, Lake Tahoe, California, May 18-21, 1997.

[4] Charles, H.R., *Fundementals of Logic Design Fourth Edition*, PWS Publishing Company, 1995.

[5] Voight, B. and Wegner, I., *A Remark On Minimal Polynomilas of Boolean Functions*, CSL'88, 2nd Workshop On Computer Science Logic Proceedings, pp. 372-383, 1989.

[6] Voight, B. and Wegner, I., *Minimal Polynomilas for the Conjunction of Functions on Disjoint Variables Can Be Very Simple*, Information and Computation 83, pp. 65-79, 1989.

[7] Mishchenko, A., Steinbach, B., Perkowski, M., *An Algorithm for Bi-Decomposition of Logic Functions*, Design Automation Conference, Proceedings, pp. 103-108, 2001.

[8] Espresso Manual, University of Berkeley CAD Group.

http://www-cad.eecs.berkeley.edu/Software/software.html

[9] Weste, N.H.E., Eshraghian K., Principles of CMOS VLSI Design, Second Edition, Addison-Wesley Publishing Company, 1993.

[10] Wu, T.C. An Introduction to Object Oriented Programming with Java Second edition, McGraw-Hill, 2000.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. YARATNON.JAVA

```
/**
 * Title:              YaratNon.java
 * Description:        This program is used to create n-variable base non-orthodox
 *                     functions, where n is even.
 *
 * @author:             Birol ULKER
 *
 */


import java.util.StringTokenizer;
import java.io.*;
import java.text.*;
import java.math.*;

 public class YaratNon {

         File resultFile;
         FileWriter fw;
         PrintWriter pw;
         String readName;

 public YaratNon() {

         System.out.println("Name For The Output File :))");

                 try{//I am getting the file names that contain the functions' MSOP that I
would like to suqare
                         InputStreamReader converter = new
                         InputStreamReader(System.in);
                         BufferedReader in = new BufferedReader(converter);
                         readName = in.readLine();

                 }
                 catch ( IOException e){}

                 try {
                         fw = new FileWriter(readName,true);
                         pw = new PrintWriter(fw);

                 }catch(Exception ex) {}
 }
```

```
        /*
                DATA MEMBERS

        */

    int n = 0;
    int w1;
    int k = 1;
    double numofMinterms;
    char[][] mMinterms;
    char essentialPIMinterms[][];
    char nonessentialPIMinterms[][];


        /*
                PUBLIC METHODS

        */


     public void start () {

        describeProgram ();
                try {
                        majorMinterms();
                } catch (Exception ex) {}

        fillArray();
        yazdirekrana();
        findEssentialPI();
        findNonEssentialPI();
     }

        /*
                PRIVATE  METHODS

        */

     private void describeProgram(){

        System.out.print("Creates the n-variable base non-orthodox function, n is even");
```

```
 }



 private void majorMinterms () throws Exception{

         System.out.print("# of variables for the NON-ORTHODOX function");
           try{
                 InputStreamReader converter = new InputStreamReader(System.in);
                 BufferedReader in = new BufferedReader(converter);
                 String text = in.readLine();
                 int i = NumberFormat.getInstance().parse(text).intValue();
                 n = i;
           }
          catch ( IOException e){}
          catch (ParseException pe) {}

                 for (int i = n; i >= 1; i--){
                         k = k*2;
                 }
}



private void fillArray(){
         mMinterms = new char[k][n];
                  for (int i = 0; i <= k-1; i++){
                         if (i >= 2){
                                 int sonuc = i;
                                 int sira = n-1;
                                         while ( sonuc >= 2){
                                                 int yaz = sonuc % 2;
                         if (yaz ==1){
                                 mMinterms[i][sira] = '1';//out bound
                                 sonuc = sonuc / 2;
                                 sira = sira -1;
                         }
                         else{
                                 mMinterms[i][sira] = '0';
                                 sonuc = sonuc / 2;
                                 sira = sira -1;
                         }
                 }

                         if (sonuc ==1 ){
                                 mMinterms[i][sira] = (char)49;
```

```
                                        sira = sira-1;
                                                for (int z = sira; z >= 0; z--){
                                                mMinterms[i][z] = '0';

                                                }
                                }


    }


                        if (i == 1){
                                for (int j = 0; j < n-1; j++){
                                        mMinterms[i][j]='0';
                                }
                                mMinterms[i][n-1] ='1';
                        }
                        else if(i ==0){
                                for (int h = 0; h <= n-1; h++){
                                        mMinterms[i][h] = '0';
                                }
                        }
  }
}


private void yazdirekrana (){

    for (int w = 0; w <=k-1; w++){
            for (int d = 0; d <= n-1; d++){
                  System.out.print(mMinterms[w][d]);
            }
                    System.out.println(" ");

    }
 }



private void findEssentialPI(){

            int counter = 0;
            int yenisayac =0;
            int alet = (int) Math.pow(2.0, n+1);
            essentialPIMinterms = new char [alet][n+1];
                    for (int b =0; b <= k-1; b++){
```

```
                    counter = 0;
                            for (int c = 0; c <= n-1; c++){

                                    if (mMinterms[b][c]==('1')){
                                            counter = counter +1;
                                    }
                            }

                    if ( counter > 0 & counter <= (n/2)){

                        for (int z =0; z <= n-1; z++){
                          essentialPIMinterms[yenisayac][z] = mMinterms[b][z];
                          essentialPIMinterms[yenisayac+1][z] = mMinterms[b][z];
                        }
                    essentialPIMinterms[yenisayac][n] = '0';
                    essentialPIMinterms[yenisayac+1][n] = '1';
                    yenisayac = yenisayac + 2;

                    }

     }

    yazdir2();
    w1 = essentialPIMinterms.length;

}



private void findNonEssentialPI() {

        int counter = 0;
        int yenisayac =0;
        int alet = (int) Math.pow(2.0, n+1);
        nonessentialPIMinterms = new char [alet][n+1];

                for (int b =0; b <= k-1; b++){
                        counter = 0;
                                for (int c = 0; c <= n-1; c++){

                                        if (mMinterms[b][c]==('1')){

                                                counter = counter +1;
                                         }
                                }
                    if ( counter >(n/2)){

                            for (int z =0; z <= n-1; z++){
                            nonessentialPIMinterms[yenisayac][z] = mMinterms[b][z];
                            }

                            nonessentialPIMinterms[yenisayac][n] = '1';
                            yenisayac = yenisayac + 1;

                    }

            }

  yazdir3();

}


private void yazdir2(){
        int counteryazma = 0;
        pw.println(".i " + (n + 1));
        pw.println(".o 1");
        int alet = (int) Math.pow(2.0, n +1);
        int hey  =0;

        for (int w = 0; w <= alet -1; w++){//
                counteryazma = 0;
                        for (int d = 0; d <= n; d++){
                                if(essentialPIMinterms[w][d]== 0){
                                        counteryazma = counteryazma + 1;

                                }

                        }

                if(counteryazma != n+1){
                        for (int k = 0; k<= n; k++){
                                pw.print(essentialPIMinterms[w][k]);
                        }
                        pw.println(" 1");
                }
        }


 }
```

```
private void yazdir3(){
        int counteryazma1=0;
        int alet = (int) Math.pow(2.0, n+1);
                for (int w = 0; w <= alet -1; w++){
                        counteryazma1=0;
                        for (int d = 0; d <= n; d++){
                                if (nonessentialPIMinterms[w][d]==0){
                                        counteryazma1 = counteryazma1 +1;
                                }

                        }
                        if(counteryazma1 != n+1){
                                for (int s = 0; s <= n; s++){
                                        pw.print(nonessentialPIMinterms[w][s]);
                                }
                                pw.println(" 1");
                        }

                }
        pw.println(".e");
        pw.close();
 }



public static void main (String[] args)  {

   YaratNon yaratNon = new YaratNon();
   yaratNon.start();
 }


}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. FAMILY.JAVA

```
/**
 * Title:              Family.java
 * Description:        This program is used to create all possible functions from the
 *                     minterms set that consists of base functions minterms and
 *                     the minterms that are provided by user. Program substitute 1s
 *                     and 0s to the provided minterms to create all possible functions
 *                     from the minterms set.
 *
 *
 * @author:             Birol ULKER
 *
 */


import java.util.StringTokenizer;
import java.io.*;
import java.text.*;
import java.math.*;
import javax.swing.*;
import java.util.*;


public class Family {
   String readName;
   String writeName;
   String newMinterm;
   File readFile;
   FileReader fr;
   BufferedReader in;
   File resultFile;
   FileWriter fw;
   PrintWriter pw;
   Vector v;
   String line;
   char[][] mMinterms;
   int k = 1;
   int n = 0;


  public Family() {

        v = new Vector();
        System.out.println("Which file u want to read ");
```

```
            try{
                  InputStreamReader converter = new InputStreamReader(System.in);
                  BufferedReader in = new BufferedReader(converter);

               readName = in.readLine();
         }
         catch ( IOException e){}

      System.out.println("Name of the OUTPUT file: ");

        try{
          InputStreamReader converter = new InputStreamReader(System.in);
          BufferedReader in = new BufferedReader(converter);
          writeName = in.readLine();
        }
       catch ( IOException e){}


  }


public void start () {

  System.out.println("Stars the program");
}




 public void getMinterms(){

     do{
          System.out.println("enter the minterms you want to try ");
          try{
                  InputStreamReader converter = new InputStreamReader(System.in);
                  BufferedReader in = new BufferedReader(converter);
                  newMinterm = in.readLine();
          }
          catch ( IOException e){}
          v.add(newMinterm);

     }while ( !newMinterm.equals("end"));

v.remove(v.size()-1);
fillArray();
  }
```

```
private void fillArray(){

          n = v.size();
                    for (int i = n; i >= 1; i--){
                              k = k*2;
                    }

          mMinterms = new char[k][n];

          for (int i = 0; i <= k-1; i++){
                    if (i >= 2){
                              int sonuc = i;
                              int sira = n-1;
                                        while ( sonuc >= 2){
                                                  int yaz = sonuc % 2;
                                                   if (yaz ==1){
                                                            mMinterms[i][sira] = '1';
                                                            sonuc = sonuc / 2;
                                                            sira = sira -1;
                                                  }
                                                  else{
                                                            mMinterms[i][sira] = '0';
                                                            sonuc = sonuc / 2;
                                                            sira = sira -1;
                                                  }
                                        }

                              if (sonuc ==1 ){
                                        mMinterms[i][sira] = (char)49;
                                        sira = sira-1;
                                                  for (int z = sira; z >= 0; z--){
                                                            mMinterms[i][z] = '0';
                                                  }
                              }


                    }

                    if (i == 1){
                              for (int j = 0; j < n-1; j++){
                                        mMinterms[i][j]='0';
                              }
                              mMinterms[i][n-1] ='1';
                    }
                    else if(i ==0){
                            for (int h = 0; h <= n-1; h++){
                                    mMinterms[i][h] = '0';
                            }
                    }
            }

    startInsertnewMinterms();
  }


  public void startInsertnewMinterms(){
     int counter  =0;
     readFile = new File(readName);
     boolean isim = true;

     for (int i = 0; i <= k-1; i++){
            counter = counter+1;
            StringBuffer str = new StringBuffer(writeName);
            try {
                    fw = new FileWriter(writeName,true);
                    pw = new PrintWriter(fw);
                    fr = new FileReader(readFile);
                    in = new BufferedReader(fr);
            }catch(Exception ex) {}

              for (int k = 0; isim ; k++){
                   if ((int)str.charAt(k)>=48 && (int)str.charAt(k)<= 57 ){
                            int j = k-1;
                            System.out.println(j);
                            writeName =  str.substring(0,j+1)+counter+".es";
                            isim = false;
                    }
              }
          isim = true;
                    try{
                            do{
                                    line = in.readLine();
                                    if ( !line .equals(".e")){
                                            pw.println(line);

                                    }
                            }while (!line .equals(".e"));

                    }catch(Exception e){}
                    try{
                            in.close();
                    }catch(Exception e){}


          for (int j = 0; j<= n-1; j++){
             if (mMinterms[i][j]=='1'){
                    pw.println(v.elementAt(j));
             }
          }

pw.println(".e");
pw.close();
   }

}


public static void main (String[] args){

   Family f = new Family();
   f.start();

    }

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APENDIX C. SONKARAR.JAVA

```
/**
 * Title:              SonKarar.java
 * Description:        Determines the types of the functions, orthodox or non-orthodox,
 *                     Espresso minimizes the functions that are created by Family.java
 *                     or Carpimtablosu.java. SonKarar.java uses Espresso’s output
 *                     files for these functions as input files.
 *
 *
 * @author             Birol ULKER
 */


import java.util.StringTokenizer;
import java.io.*;
import java.text.*;
import java.math.*;
import javax.swing.*;
import java.util.*;



public class SonKarar {

   String readName;
   String writeName;
   String  numFiles;
   String  checkCharNum;
   File readFile;
   FileReader fr;
   BufferedReader in;
   File resultFile;
   FileWriter fw;
   PrintWriter pw;
   int n;
   int birol;


  public SonKarar() {

     System.out.println("how many files Do I need to check BOSS :))");

    try{
         InputStreamReader converter = new InputStreamReader(System.in);
         BufferedReader in = new BufferedReader(converter);

         numFiles = in.readLine();
         n = Integer.valueOf( numFiles).intValue();
    }
   catch ( IOException e){}

      System.out.println("OK TELL ME THE NAME OF THE FILES BOSS :))");

   try{
         InputStreamReader converter = new InputStreamReader(System.in);
         BufferedReader in = new BufferedReader(converter);
         readName = in.readLine();

    }
   catch ( IOException e){}


      System.out.println("RESULT FILE NAME BOSS :))");

   try{
         InputStreamReader converter = new InputStreamReader(System.in);
         BufferedReader in = new BufferedReader(converter);
         writeName = in.readLine();

    }
   catch ( IOException e){}

System.out.println("WHICH CHAR SHOULD BE CHECKED BUS :))");

     try{
         InputStreamReader converter = new InputStreamReader(System.in);
         BufferedReader in = new BufferedReader(converter);
        checkCharNum = in.readLine();
        birol = Integer.valueOf( checkCharNum).intValue();
    }
   catch ( IOException e){}


}



   public void start(){

         int counter  =0;
```

```
try {
        fw = new FileWriter(writeName,true);
        pw = new PrintWriter(fw);
}catch(Exception ex) {}
boolean isim = true;

        for (int i = 0; i <= n-1 ; i++){
                counter = counter+1;
                StringBuffer str = new StringBuffer(readName);

        try {
                fr = new FileReader(readName);//okuma icin
                in = new BufferedReader(fr);//okuma icin
        }catch(Exception ex) {}



         for (int k = 0; isim ; k++){
              if ((int)str.charAt(k)>=48 && (int)str.charAt(k)<= 57 ){
                 int j = k-1;
                 System.out.println(j);
                 readName =  str.substring(0,j+1)+counter+".out";
                 isim = false;
              }
          }

isim = true;


   String line;
   int sayac = 0;
         try{
                while((line = in.readLine()) != null){
                        if (line.length() > birol){
                                if (line.charAt(birol)== '1'){
                                sayac = sayac +1;
                                }
                        }

                }

        }catch(Exception e){}


 if (sayac > 1){
     pw.println("NON-ORTHODOX ");
```

```
            }
          else{
                pw.println("ORTHODOX FUNCTION !! ORTHODOX FUNCTION !!
ORTHODOX FUNCTION !!");
            }
        }
        pw.close();


}



public static void main (String[] args){

   SonKarar sonKarar = new SonKarar();
    sonKarar.start();

  }

}
```

APPENDIX D. CARPIMTABLOSU.JAVA

```
/**
 * Title:              CarpimTablosu.java
 * Description:        This program creates f^2. Logically AND the given n-variable
 *                     with itself as follows;
 *                     f^2= f(X)f(Y), where X∩Y=0
 *
 * @author             Birol ULKER
 */


import java.util.StringTokenizer;
import java.io.*;
import java.text.*;
import java.math.*;
import javax.swing.*;
import java.util.*;


public class CarpimTablosu {

        String readName;
        String inPutNum;
        String writeName;
        String  numFiles;
        String line;
        File readFile;
        FileReader fr;
        BufferedReader in;
        File resultFile;
        FileWriter fw;
        PrintWriter pw;
        int n;
        Vector v;

  public CarpimTablosu() {

      System.out.println("how many functions  Do I need to MULTIPLY BOSS :))");

        try{
                InputStreamReader converter = new InputStreamReader(System.in);
                BufferedReader in = new BufferedReader(converter);
                numFiles = in.readLine();
                n = Integer.valueOf( numFiles).intValue();
        }
        catch ( IOException e){}



      System.out.println(".i for new files :))");

        try{
                InputStreamReader converter = new InputStreamReader(System.in);
                BufferedReader in = new BufferedReader(converter);
                inPutNum = in.readLine();

        }
        catch ( IOException e){}

    System.out.println("OK TELL ME THE FUNCTIONS YOU WANT TO SQUARE
BOSS :))");

        try{
                InputStreamReader converter = new InputStreamReader(System.in);
                BufferedReader in = new BufferedReader(converter);
                readName = in.readLine();
        }
        catch ( IOException e){}

  System.out.println("RESULT FILE NAME BOSS :))");
        try{
                InputStreamReader converter = new InputStreamReader(System.in);
                BufferedReader in = new BufferedReader(converter);
                writeName = in.readLine();

         }
        catch ( IOException e){}
}



   public void start(){

        int counter1  =0;
        int counter2  =0;
        boolean isim1 = true;
        boolean isim2 = true;
        boolean isim3 = true;

                for (int i = 0; i <= n-1 ; i++){
```

```
                counter1 = counter1+1;
                counter2 = counter2+1;
                StringBuffer str1 = new StringBuffer(readName);
                StringBuffer str2 = new StringBuffer(writeName);
                v = new Vector();
        try {
            fr = new FileReader(readName);//okuma icin
            in = new BufferedReader(fr);//okuma icin
        }catch(Exception ex) {}

        try {
           fw = new FileWriter(writeName,true);
           pw = new PrintWriter(fw);
        }catch(Exception ex) {}


         for (int z = 0; isim1 ; z++){
              if ((int)str1.charAt(z)>=48 && (int)str1.charAt(z)<= 57 ){
                 int j = z-1;
                 System.out.println(j);
                 readName =  str1.substring(0,j+1)+counter1+".es";
                 isim1 = false;
              }
          }

isim1 = true;
          for (int k = 0; isim2 ; k++){
              if ((int)str2.charAt(k)>=48 && (int)str2.charAt(k)<= 57 ){
                 int h = k-1;
                 System.out.println(h);
                  writeName =  str2.substring(0,h+1)+counter2+".es";
                  isim2 = false;
              }
            }

 isim2 = true;

          try{
                do{
                        line = in.readLine();
                        if ( line.charAt(0)!='.'){
                                StringBuffer str3 = new StringBuffer(line);
                                for (int f = 0; isim3 ; f++){
                                   if ((int)str3.charAt(f)== 32 ){
                                       int u = f-1;
                                       line =  str3.substring(0,u+1);
```

```
                                            isim3 = false;
                                        }
                                    }

                               isim3 = true;
                               v.add(line);

                               }
                       }while (!line .equals(".e"));

             }catch(Exception e){}

            try{
               in.close();
            }catch(Exception e){}

            pw.println(".i"+" "+inPutNum);
            pw.println(".o 1");
            pw.println(".e");
            pw.close();

    }

  }



      public static void main (String[] args){

            CarpimTablosu carpimTablosu = new CarpimTablosu();
            carpimTablosu.start();
      }

  }
```

APPENDIX E. COMPARE.JAVA

```
/**
 * Title:              Compare.java
 * Description:        This program used to determine the penalty we need to
 *                     pay when we minimize the functions by applying law
 *                     of distributivity.  Decomposable functions obtained from
 *                     outputs of CarpimTablosu.java. Espresso used to minimize
 *                     both the subunctions of the decomposable function and
 *                     decomposable function itself. Compare.java compares
 *                     the results and finds the differnee between τ(MSOP : f)^2 and
 *                     τ(MSOP : f^2).
 * @author             Birol ULKER
 */


import java.util.StringTokenizer;
import java.io.*;
import java.text.*;
import java.math.*;
import javax.swing.*;
import java.util.*;


public class Compare {

   String readName2;
   String readName;
   String writeName;
   String writeName2;
   String  numFiles;
   File readFile;
   FileReader fr;
   BufferedReader in;
   File resultFile;
   FileWriter fw;
   PrintWriter pw;
   int n;
   int org;
   Vector v1;
   Vector vOrg;
   int birol;
   String  checkCharNum;
 public Compare() {
      System.out.println("how many files Do I need to check BOSS :))");

         try{
                 InputStreamReader converter = new InputStreamReader(System.in);
                 BufferedReader in = new BufferedReader(converter);
                 numFiles = in.readLine();
                 n = Integer.valueOf( numFiles).intValue();
                 org = Integer.valueOf( numFiles).intValue();
         }
         catch ( IOException e){}

      System.out.println("OK TELL ME THE NAME OF THE FILES BOSS :))");

         try{
                 InputStreamReader converter = new InputStreamReader(System.in);
                 BufferedReader in = new BufferedReader(converter);
                 readName = in.readLine();
         }
         catch ( IOException e){}



      System.out.println("NAME OF THE FILES FOR THE ORIGINAL FUNCTIONS
BOSS :))");

         try{
                 InputStreamReader converter = new InputStreamReader(System.in);
                 BufferedReader in = new BufferedReader(converter);
                 readName2 = in.readLine();
         }
         catch ( IOException e){}


     System.out.println("FILE NAME BOSS for the squared ones  I TELL YOU NON-
ORT. OR ORT.:)) ");
         try{
                 InputStreamReader converter = new InputStreamReader(System.in);
                 BufferedReader in = new BufferedReader(converter);
                 writeName = in.readLine();
         }
         catch ( IOException e){}
  System.out.println("RESULT FOR THE SUBTRACTION :))");

         try{
                 InputStreamReader converter = new InputStreamReader(System.in);
                 BufferedReader in = new BufferedReader(converter);
                 writeName2 = in.readLine();
         }
         catch ( IOException e){}

    System.out.println("WHICH CHAR SHOULD BE CHECKED BUS :))");

         try{
                 InputStreamReader converter = new InputStreamReader(System.in);
                 BufferedReader in = new BufferedReader(converter);
                 checkCharNum = in.readLine();
                 birol = Integer.valueOf( checkCharNum).intValue();
         }
         catch ( IOException e){}
}




public void start(){

    v1 = new Vector();
    int counter  =0;
         try {
                 fw = new FileWriter(writeName,true);
                 pw = new PrintWriter(fw);
         }catch(Exception ex) {}
    boolean isim = true;
    int x = 0;
         for (int i = 0; i <= n-1 ; i++){
              counter = counter+1;
              StringBuffer str = new StringBuffer(readName);

               try {

                  fr = new FileReader(readName);
                  in = new BufferedReader(fr);
               }catch(Exception ex) {}

                 for (int k = 0; isim ; k++){

                      if ((int)str.charAt(k)>=48 && (int)str.charAt(k)<= 57 ){
```

```
                              int j = k-1;
                              readName =  str.substring(0,j+1)+counter+".out";
                              isim = false;
                        }
                    }

        isim = true;
        String line;
        String cikti;
        int sayac = 0;
        StringBuffer strOutPutSayisi;

            try{
                while((line = in.readLine()) != null){

                    strOutPutSayisi = new StringBuffer(line);
                               if (strOutPutSayisi.charAt(1) =='p'){
                                  cikti =
                       strOutPutSayisi.substring(2,strOutPutSayisi.length());

                                  v1.add(cikti);

                                }

                               if (line.length() > birol){

                                      if (line.charAt(birol)== '1'){
                                      sayac = sayac +1;
                                      }
                               }

                      }

            }catch(Exception e){}


         if (sayac > 1){
             pw.println("NON-ORTHODOX ");
          }
          else{
             pw.println("ORTHODOX FUNCTION !! ORTHODOX FUNCTION !!
ORTHODOX FUNCTION !!");
          }
         }
         pw.close();
        ikinciOkuma();
 }


public void ikinciOkuma(){

   vOrg = new Vector();
   int counter  =0;
   boolean isim = true;
   int q = 0;

   for (int i = 0; i <= org-1 ; i++){

        counter = counter+1;
        StringBuffer str = new StringBuffer(readName2);

             try {

                     fr = new FileReader(readName2);
                     in = new BufferedReader(fr);
              }catch(Exception ex) {}



                      for (int k = 0; isim ; k++){

                          if ((int)str.charAt(k)>=48 && (int)str.charAt(k)<= 57 ){
                             int j = k-1;
                             readName2 =  str.substring(0,j+1)+counter+".out";
                             isim = false;
                          }
                      }

                 isim = true;


      String line2;
      String cikti2;
      int sayac = 0;
      StringBuffer strOutPutSayisi;

              try{
                   while((line2 = in.readLine()) != null){
                        strOutPutSayisi = new StringBuffer(line2);
                                    if (strOutPutSayisi.charAt(1) =='p'){
                                         cikti2 =
                                   strOutPutSayisi.substring(2,strOutPutSayisi.length());

                                          vOrg.add(cikti2);
                                     }
                        }
                }catch(Exception e){}
        }
        Subtraction();

   }


   public void  Subtraction(){
        double result;
        int outputv1;
        int outputvOrg;
         try {

                fw = new FileWriter(writeName2,true);
                pw = new PrintWriter(fw);
        }catch(Exception ex) {}

        for (int filiz = 0; filiz < v1.size(); filiz++){
             outputv1 = Integer.valueOf( v1.elementAt(filiz).toString().trim()).intValue();
             outputvOrg = Integer.valueOf(
vOrg.elementAt(filiz).toString().trim()).intValue();
             double kim = Math.pow((double)outputvOrg,2.0);
             result =  kim-outputv1;
              pw.print(result);
              pw.println("   "+filiz);
           }
        pw.close();
}


public static void main (String[] args){

  Compare compare = new Compare();
   compare.start();
   }



}
```

APPENDIX F. ESPRESSO2.JAVA

```
/**
 * Title:              Espresso2.java
 * Description:        This program is used to logically AND two functions
 * @author             Birol ULKER
 */


import java.util.StringTokenizer;
import java.io.*;
import java.text.*;
import java.math.*;


public class espresso2{

  File resultFile;
  FileWriter fw;
  PrintWriter pw;
  InputStreamReader converter;
  BufferedReader in;

  StringTokenizer st;
  String function;
  String[] functionOne;
  String[] functionTwo;
  String[] resultFunction;
  String[] variables;
  int numberOfVariables;
  int numberOfElements;
  int[][] intVariables;
  boolean[] boolVariables;
  boolean append = false;

  public espresso2() {
  }



   static public void main(String[] args) throws Exception
   {
     boolean exit = false;
     char test;
     espresso2 example = new espresso2();
```

```
      example.InitializeStreams();

          while (!exit)       {

                    example.GetInputs();
                    example.ConstructResult();
                    example.Display();
                    example.CalculateFunctionVariables();
                    example.CalculateFunction();
                    System.out.println(" ");
                    System.out.print("to continue enter any character to exit enter h:");
                    test = (char) System.in.read();
                              if (test == 'h') {

                                        exit = true;
                              }
                    System.in.skip(2);
                    }
    }



    public void InitializeStreams() throws Exception{

      char test;
      converter  = new InputStreamReader(System.in);
      in         = new BufferedReader(converter);

      System.out.println("Please enter the path for the result file:");
      resultFile = new File(in.readLine());

      System.out.println("to append the results to the file enter 1 if not any other
character:");
      test = in.readLine().charAt(0);
          if (test == '1') {

              append = true;
          }

      fw         = new FileWriter(resultFile.getName(), true);
      pw         = new PrintWriter(fw);


    }
```

```
public void GetInputs() throws Exception{

    numberOfElements = 0;

            for (int i = 0; i < 3; i++){

                    if (i == 0) {

                    System.out.println("Please enter the first function");
                    function = in.readLine();
                    st = new StringTokenizer(function, "+");
                    numberOfElements = st.countTokens();
                    functionOne = new String[numberOfElements];

                            for(int j = 0; j < numberOfElements; j++){

                                    functionOne[j] = st.nextToken();
                            }
                    }
                    else if(i == 1) {

                            System.out.println("Please enter the second function");
                            function = in.readLine();
                            st = new StringTokenizer(function, "+");
                            numberOfElements = st.countTokens();
                            functionTwo = new String[numberOfElements];

                              for(int j = 0; j < numberOfElements; j++){

                                functionTwo[j] = st.nextToken();
                              }
                    }
                    else {
                            System.out.println("Please enter variables (ex: xyab...)");
                            String text = in.readLine();
                            char[] temp = new char[1];
                            numberOfVariables = text.length();
                            boolVariables = new boolean[numberOfVariables];
                                            intVariables = new int[(int)Math.pow(2.0,
            numberOfVariables)][numberOfVariables];
                            variables = new String[numberOfVariables];

                            for (int k = 0; k < numberOfVariables; k++){

                                    temp[0] = text.charAt(k);
```

```
                                   variables[k] = new String(temp);
                                  }
                          }
         }
  }




public void ConstructResult(){

  numberOfElements = functionOne.length * functionTwo.length;
  resultFunction = new String[numberOfElements];
  int index = 0;
      for (int i = 0; i < functionTwo.length; i++){

          for (int j = 0; j < functionOne.length; j++){

                resultFunction[index] = functionOne[j].concat(functionTwo[i]);
                index++;
          }
      }
}


public void Display(){

  System.out.println("The multiplication of given two function is:");
  pw.println(" ");
  pw.println(" ");
  pw.println("The multiplication of given two function is:");
      for(int i = 0 ; i < resultFunction.length; i++){

                System.out.print(resultFunction[i]);
                pw.print(resultFunction[i]);
                          if (i != resultFunction.length - 1) {
                                    System.out.print(" + ");
                                    pw.print(" + ");
                          }
      }
 pw.println(" ");
}

 public void CalculateFunctionVariables(){

      int number = (int) Math.pow(2.0, numberOfVariables);
```

```
        int n = 0;

                for (int i = 0; i < number; i++){

                        n = i;

                                for (int j = (numberOfVariables - 1); j >= 0; j--){

                                        if (n >= 0) {

                                                intVariables[i][j] = n % 2;
                                        }

                                        n = ((n - (n % 2)) / 2);
                                        }
                                }
    }



public void CalculateFunction() throws Exception{

      boolean[] variables = new boolean[numberOfVariables];
      boolean functionResult = false;

      System.out.println(" ");
      pw.println(" ");
      System.out.println("The result is calculated as follow: ");
      pw.println("The result is calculated as follow: ");
         for (int p = 0; p < numberOfVariables; p++){

                 System.out.print(this.variables[p] + "");
                 pw.print(this.variables[p] + "");
         }

      pw.println(" ");
      pw.println(".i " + numberOfVariables);
      pw.println(".o 1");

         for (int i = 0; i < ((int) Math.pow(2.0, numberOfVariables)); i++){

                 for (int j = 0; j < numberOfVariables; j++){

                         if (intVariables[i][j] == 0) {

                                 variables[j] = false;
                        }
                        else{

                                variables[j] = true;
                        }
                }

    functionResult = Calculate(variables);

                if (functionResult == true) {

                        System.out.println(" ");
                        pw.println(" ");
                                for (int k = 0; k < numberOfVariables; k++){

                                        System.out.print(intVariables[i][k]);
                                        pw.print(intVariables[i][k]);
                                }
                        pw.print(" 1");
                }
  }
  pw.println(" ");
  pw.println(".e");
  fw.close();
}


public boolean Calculate(boolean[] booleanVariables) {

  boolean result    = true;

      for (int i = 0; i < resultFunction.length; i++){

                boolean[] value = new boolean[(resultFunction[i].length() / 2)];

                        for (int j = 0; j < (resultFunction[i].length() / 2); j++){

                                result = true;
                                char[] tempTest = new char[1];
                                tempTest[0] = resultFunction[i].charAt(2 * j);
                                String test = new String(tempTest);

                                        if (resultFunction[i].charAt((2 * j) + 1) == '1') {

                                                for (int p = 0; p < variables.length; p++){
                                                    if (test.equals(variables[p])) {

                                                        value[j] = booleanVariables[p];
                                                    }
                                                }
                                        }
                                        else{

                                         for (int p = 0; p < variables.length; p++){

                                                if (test.equals(variables[p])) {

                                                        value[j] = !booleanVariables[p];
                                                }
                                        }
                                }
                        }

                for (int m = 0; m < value.length; m++){

                        result = result & value[m];
                }

                if (result == true) {

                        return true;
                }
                }
            return false;
        }

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Deniz Kuvvetleri Komutanligi
   Personel Dairesi Baskanligi
   Bakanliklar Ankara, Turkiye

3. Deniz Harp Okulu Komutanligi
   Kutuphane
   Tuzla Istanbul, Turkiye

4. Dudley Knox Library
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

5. Chairman, Department of Electrical and Computer Engineering
   Code ECE
   Naval Postgraduate School
   Monterey, California 93943

6. Prof. Jon T. Butler, Code ECE/CS
   Code ECE
   Naval Postgraduate School
   Monterey, California 93943

7. Prof. Herschel H. LOOMIS
   Code ECE
   Naval Postgraduate School
   Monterey, California 93943

8. Dz.Y.Muh.Kd.Bnb. Onur Oral Ulker
   EH. Tek. Des.S.M.
   Genelkurmay Baskanligi
   Ankara, Turkiye

9. LTJG Birol Ulker
   Arpaemini Mah. Pazartekke Cikmazi Sok.
   No: 34 D: 10 Sehremini
   Istanbul, Turkiye